Contents

List of Figures

Preface

XPPAUT is a tool for simulating, animating, and analyzing dynamical systems. The program evolved from a DOS program that was originally written so that John Rinzel and I could easily illustrate the dynamics of a simple model for an excitable membrane. The DOS program, PHASEPLANE, became a commercial project and was used for many years by a number of patient folks. In the early 1990's I spent a month in a beautiful office at the Mathematical Sciences Research Institute as part of a Mathematical Biology program. During the evenings, I ported the DOS program to X-Windows on a UNIX environment while enjoying the sunset and listening to the same cassette tape over and over. (I forget what it was.) The program has evolved a great deal from those early years and is available at no cost to anyone who wishes to download it. I have also successfully compiled the X version to run under various 32-bit flavors of Windows and also the new Mac OS-X.

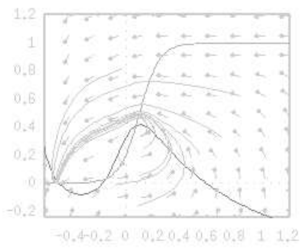
I have added lots of integrators and tools as well as my own idiosyncratic interface to the amazing continuation package AUTO. Most things that you might want to do that concern dynamics – either discrete or continuous – can probably be done with *XPPAUT* if you know a few of the tricks. That is the point of this book; I suspect many users do not take full advantage of the features of the program. This is mostly my fault as the user manual that is distributed with the program, while comprehensive in its description of all the features, is hopelessly baroque in its organization.

Why should anyone want to use *XPPAUT* ? There are plenty of packages that will integrate differential equations for you. Many people use MATLAB, MAPLE, or MATHEMATICA to study and analyze dynamical systems. These are all general purpose packages that have the capability to do most everything that is described in this book. However, both of the symbolic packages are extremely slow when it comes to numerically solving differential equations. Furthermore, there is not much flexibility in the choice of integration methods and the integration is not done interactively. That is, you cannot see the progress of the solution until it is computed. Standard qualitative tools such as direction fields and nullclines require additional packages or writing by hand. MATLAB has great flexibility and can even integrate differential equations with discontinuities such as the integrate-and-fire equations. However, the numerical integration is generally slower than can be achieved with *XPPAUT* . None of the packages offers an interface to AUTO, the main reason that some people use *XPPAUT* . The syntax for setting up differential equations is pretty simple compared to the other programs. Finally, *XPPAUT* is free – no license demons crashing once a year, no guilt copying to another computer, and the source code is always there for the taking.

How to use this book. This book is written to be used by either a researcher or modeler who wants to simulate and analyze her system or by students as an adjunct to a modeling class or a class in differential equations. I have used it in many such courses both at the level of sophomore engineering students up through graduate students in a dynamical systems class. I have used *XPPAUT* in applied courses for students in neuroscience and physiology. The present book contains many examples and many exercises. Along the way, I hope that it can aid in teaching certain concepts in the analysis of the behavior of differential equations. Most of the problems and examples are taken from research papers. The emphasis, I am afraid, is skewed toward biological applications as that is what I do.

If all you want to do is solve differential equations and graph the solutions, then most of what you want can be found in chapter 3. Suggestions for how you can use *XPPAUT* in a classroom setting are found in chapter 4. Research problems involve a more complete set of tools. Chapter 5 introduces functional and stochastic differential equations while chapter 6 shows you how to discretize partial-differential equations and solve these. Boundary-value problems are also covered in this chapter. Tricks and special classes of differential equations are described in 9. Chapter 7 introduces bifurcation theory and the use of the AUTO interface in *XPPAUT* . Chapter 8 shows you how to make animations with the built-in animator and 9 shows other ways to make animations.

Acknowledgments I have benefitted a great deal from the many users of various versions of the program. To those one or two of you who sent me a note about how useful the program is rather than what new features you wished I'd put into it or which ones don't work, I salute you. To the others, well, I reluctantly thank you as your comments motivated new features and bugs. Mostly, I would thank John Rinzel and Artie Sherman for being guinea pigs for many versions of the program that have appeared throughout the years. Finally, I want to thank my wife, Ellen, for her patience, and the boys, Kyle and Jordan, without whom this book would have a 2000 copyright.



Chapter 1

Installation

Like all Holme's reasoning the thing seemed simplicity itself when it was once explained.
–Arthur Conan Doyle, *The Stock-broker's Clerk*

Installation of *XPPAUT* is done either by downloading the source code and compiling it or downloading one of the binary versions. I will give sample installations for UNIX, Windows, and MacOS X. If you are totally clueless at compiling source code, it is best to either have your system administrator install it for you or download a precompiled binary for your computer. There are compiled versions available for Linux, SUN, HP, Windows, and Mac OSX.

1.1 Installation on UNIX

1.1.1 Installation from the source code

Create a directory called `xppaut` and change to this directory by typing:

```
mkdir xppaut
cd xppaut
```

Step 1. Download the compressed tarred source code `xppaut_latest.tar.gz` into this directory from one of the two URLs:

- http://www.math.pitt.edu/*asymp*bard/xpp/xpp.html
- http://www.cnbc.cmu.edu/≍bard/files.html

Step 2. Uncompress and untar the archive:

```
gunzip xppaut_latest.tar.gz
tar xf xppaut_latest.tar
```

This will create a series of files and subdirectories.

Step 3. Type

```
make
```

and lots of things will scroll by including occasional warnings (that you can safely ignore). If you get no errors, then you probably have succeeded in the compilation. If the compilation stops very quickly, then you probably you will have to edit the Makefile according to the architecture of your computer. Look at the README file and the Makefile which has suggestions for many platforms.

Step 4. If you successfully have compiled the program, then you should have a file `xppaut` in your directory. To see, type

```
ls xppaut
```

If you see something like `xppaut*` listed then you have succeeded. If you don't see this, then the compilation was unsuccessful. Consult the README file for a variety of possible fixes. Also, there are many comments in the Makefiles that are included with the package. I have not yet found a computer on which I cannot compile the program. Common problems are the wrong path to the X Windows libraries, nonexistence of `ranlib` among others.

Step 5. Once you have compiled it, just move the executable to someplace in your path. (The usual is `/usr/local/bin` but you must have root privileges to do this.) *XPPAUT* needs no environment information.

1.1.2 Installation from binaries

Some binaries are available at one or both of the above URLs. You should download these as well as the source code above. The source code has many examples and the *XPPAUT* reference manual. Download a binary, e.g., `xppaut4.6_hpux.gz` and uncompress it with the command `gunzip xppaut4.6_hpux.gz` and copy it to the desired directory. The binaries are missing things like the example files and the documentation. Download the source code to get these.

1.1.3 Additional UNIX setup

In some systems, the zooming and cursor movement does not always work properly.
In these systems, you want to call *XPPAUT* with an additional command line argument, e.g.,

```
xppaut -xorfix file.ode
```

This will usually fix these problems. By default, *XPPAUT* comes up with only the main window visible. (You can always make the other windows visible by clicking on the top row of buttons on the main window.) You may want to have *XPPAUT* come up with all the windows visible To do this, add the command line argument,

-allwin. You can use the alias command in your shell to call *XPPAUT* with these command line arguments. Alternatively, create a text file called, e.g., myxpp with the following line in it:

```
\usr\local\bin\xppaut %1 %2 -allwin -xorfix
```

Save the file and make it executable by typing chmod +x myxpp. Now if you call myxpp, it will have the two command-line options enabled.

1.2 Native MS Windows NT/95/98/2000

Just download the program winpp.zip into a folder, say **wpp** and then use Winzip or a similar program to unzip the file. Create a shortcut to winpp. This version does not have all the features of the full version. Furthermore, the interface is quite different. Most of the equation files will work for this version and most of the standard features are extant. There is a binary X version for Windows which is identical to the full UNIX version and I recommend that you use that instead as this book describes the X version. (See the next section.)

1.3 X-windows version on Windows.

This is the recommended way to run the program in the Windows environment. It is only slightly more difficult to install. It does not use the Windows API, but works identically to the UNIX version. **NOTE.** If you have only used an X-windows emulator to log into another machine, this may be a bit of a surprise. You can run local programs which are properly compiled X-windows programs right on your PC with the X-emulator running. *You do not have to be on a network to run this program on your Windows PC.*

Before you download *XPPAUT* on a Windows machine, you should have X-windows emulator. There are a number of them available at a cost or as demos. There are at least three that are very inexpensive:

X-WinPro: The demo version runs for 30 minutes at a time and the full version is $90. URL: http://www.labf.com/index.html

X-Win32: The demo version runs for 120 minutes at a time. I use this product at home. Prices range from $50 for students to $200 for corporations. URL: http://www.starnet.com/productinfo/

MI/X: This is the smallest and has the fewest features. The demo lasts for 15 days. The cost is $25. URL:http://www.microimages.com/

They are all pretty simple to install and take up very little disk space. Many universities have site licenses for X-servers such as Exceed (see their site:

```
http://www.hummingbird.com/products/nc/exceed
```

Here are the steps to install *XPPAUT* in Windows:

Step 1. Create a folder `tmp` that will be a temporary directory. Create another folder called `xpp`.

Step 2. If you have an X-windows emulator already, then skip this step. Otherwise, you should install one of the above Xservers from the `tmp` directory or the desktop. I have an old version of the MI/X server available to download. If you want to try it, here is how:

- Download the two files into the `tmp` directory: `runme1st.exe` and `file000.bin`.
- Run the program `runme1st.exe` to install an X windows server onto your computer. This server only needs a few megabytes of disk space so it is pretty small. Test the installation by clicking on the START menu and running the program found under TNT Lite.

Step 3. Download the file `xpp4win.zip` into the folder `xpp`. Unzip this with the Winzip utility. There will be a number of files including `xppaut.exe`. Note that there are two dynamic link libraries (DLL's) in the zipped file, so, if you want to move `xppaut.exe` to a different directory, you should move the DLLs there as well. To make it available everywher, you can copy `xppaut.exe`, `cygwin1.dll`, and `libX11.dll` into your Programs directory or any other directory in your path.

Step 4. Test your download.

Step A. Start your X-server.

Step B. Open a MS-DOS prompt from the START menu. Change to the `xpp` directory. (In Windows 2000, this is called Command Prompt. It is available off the Start/Applications menu. If you cannot find it, click on `Run` and type in `command.com`.)

Step C Now you have to tell X where to send the display.

If you are on a network. Type `set DISPLAY=mypc:0.0` where `mypc` is the name of your PC on the network.

If you are not on a network. Type `set DISPLAY=127.0.0.1:0.0`

Note that even on a network, the second command usually works.

Step D You are now ready to run. Type `xppaut lecar.ode` and *XPPAUT* should fire up in the X-window. If not, then check that you have started the X server and set the DISPLAY correctly. Note that, if you get an error `Can't open display` then you should try to find out the name of your PC as that is probably the problem. Another possibility is that your X server won't let your PC host the display. Look for something that allows you to set HOSTS in your X-server and set the host to your display name.

Step E. If successful, exit *XPPAUT* by clicking on the `File` and then the `Quit` entry and answer Yes.

NOTE. My home computer is not on a network, so I have just created a batch file `xpp.bat` and included in it a line that sets the DISPLAY for me:

```
set BROWSER=c:\Program Files\Netscape\Communicator\Program\netscape.exe
set XPPHELP=c:\xpp\help\xpphelp.html
set DISPLAY=127.0.0.1:0.0
set HOME=c:\xpp
C:\xpp\xppaut %1 %2 %3
```

Note this also sets some other environmental variables.

1.4 Installation on MacOSX

Installation on Macintosh computers running OSX is possible by downloading the source code for *XPPAUT* and then compiling it using the software development tools provided for the new OS. In addition, you will need to download the X development libraries to compile it. The following steps were helpfully provided to me by Chris Fall and James Sneyd. I have managed to test this on one laptop and everything seems to work. A Mac OSX binary can be found on the web site if you don't want to compile it yourself.

1. Make sure you get and install the full Developer Kit for Mac OSX. This is how you get the cc compiler.

2. Install XFree86 on OSX. Download from

   `ftp://ftp.xfree86.org/pub/XFree86/4.1.0/binaries/Darwin-ppc/`

   Make sure (no matter what the Install file says) that you also get the `Xprog.tgz` bundle. You need it.

3. Get the *xpp* source code and put it in a directory of your choice. I'll assume you've called it xpp. Untar the archive.

4. Make the following changes in the MAC system directories. (I think this is a bug in their header files.)

   - copy `/usr/include/dirent.h` to your xpp directory. I'll assume you've called it `dirent.h` locally.
   - copy `/usr/include/sys/dirent.h` to your xpp directory (giving it a new name obviously. I called it `sysdirent.h`).
   - In the file `read_dir.c` change the `#include <dirent.h>` statement to call your local copy of `dirent.h`, not the one in `/usr/include`.
   - In your local copy of `dirent.h`, change the `#include <sys/dirent.h>` statement to call your local copy of `sysdirent.h`.
   - In your local copy of sysdirent.h, change the lines:

```
    u_int32_t d_fileno;          /* file number of entry */
    u_int16_t d_reclen;          /* length of this record */
    u_int8_t  d_type;          /* file type, see below */
    u_int8_t  d_namlen;      /* length of string in d_name */
```

to the new lines:

```
    unsigned long d_fileno;         /* file number of entry */
    unsigned short d_reclen;        /* length of this record */
    unsigned char d_type;        /* file type, see below */
    unsigned char d_namlen;     /* length of string in d_name */
```

(These occur in the `struct dirent` declaration) and save the file.

5. In the Makefile use the following options

```
CC= cc
CFLAGS=  -O -DAUTO -DCVODE_YES  -I/usr/X11R6/include
LDFLAGS= -L/usr/X11R6/lib
AUTLIBS= -lf2c -lX11 -lm
LIBS= -lX11 -lm
OTHERLIBS= libcvode.a libf2cm.a
```

Note that in the subdirectories, `cvodesrc` and `libI77` make sure that `CC=cc`.

Then in the main directory, type `make` and everything should go fine. The rest of the story is like the UNIX installation.

1.5 Environment variables.

XPPAUT makes use of certain environment variables. In UNIX, these are set in files such as `.bashrc`. In order to use the online help, *XPPAUT* needs to know the starting help file. For example, on my Linux computer, this is `/home/bard/xpppaut/help/xpphelp.html`. *XPPAUT* also needs to know the name of your browser since it calls the browser to display the help. Thus, in my `.bashrc` file, I have the two lines

```
export XPPHELP=/home/bard/xppnew/help/xpphelp.html
export BROWSER=/usr/bin/netscape
```

The same should work for OSX since this new Mac operating system is Unix.

In Windows, you can do the same thing using the "set" command.

```
set BROWSER=c:\Program Files\Netscape\Communicator\Program\netscape.exe
set XPPHELP=c:\xpp\help\xpphelp.html
```

I usually include all this in a batch file which sets up other parameters as well. (See below.)

1.5.1 Resource file.

The other environment variable that *XPPAUT* makes use of is the `HOME` directory. *XPPAUT* looks here for the file `.xpprc`. Each time *XPPAUT* is run, it loads the options defined in `.xpprc`. These are described in appendix B. . The resource file, `.xpprc` just contains global options that you might want to have for every ODE that you run. For example, here is a short one:

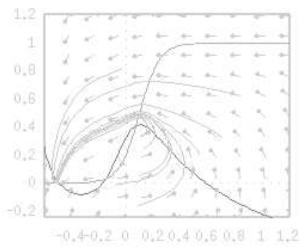
```
# my xpprc file
@ but=quit:fq
@ maxstor=50000,bell=0
@ meth=qualrk,tol=1e-6,atol=1e-6
# thats it
```

This automatically puts a quit button on the top bar, allocates 50000 storage points (instead of the default, 5000), makes the default integrator the adaptive Runge-Kutta, and turns off the bell.

In Windows, you set the "HOME" directory and in that directory create a file called `.xpprc`. It will be the same form as the UNIX version. I usually make a batch file which does everything for me automatically. I repeat my batchfile, `xpp.bat` here:

```
set BROWSER=c:\Program Files\Netscape\Communicator\Program\netscape.exe
set XPPHELP=c:\xpp\help\xpphelp.html
set DISPLAY=127.0.0.1:0.0
set HOME=c:\xpp
c:\xpp\xppaut %1 %2 %3
```

This sets the display for X and also tells *XPPAUT* where to look for the resource file, the browser, and the first page of the help file. Then it calls *XPPAUT* .



Chapter 2

A very brief tour of *XPPAUT*

Touring can make you crazy.
–Frank Zappa

In this chapter, I will show you the way to get up and running. Many of the types of problems most people need to solve can be done after going through this short session with the program. I will assume that you are at least somewhat familiar with differential equations.

I will run through two quick tutorials on using *XPPAUT* for a linear differential equation and a nonlinear differential equations. This should be sufficient for anyone to setup differential equations and solve them. For more advanced topics, the user should look at the rest of the book.

NOTES: Menu commands will appear like this `Command` and single letter keyboard shortcuts will appear like this: **A** . *Do not use the CapsLock key*; all shortcuts are lower case. Every command can be accessed by a series of keystrokes. To make sure key clicks are interpreted correctly, click on the title bar of the window for which the shortcut is intended.

2.1 Creating the ODE file.

Consider the simple linear differential equation:

$$\begin{aligned}\frac{dx}{dt} &= ax + by \\ \frac{dy}{dt} &= cx + dy \qquad (2.1)\end{aligned}$$

where a, b, c, d are parameters. We will explore the behavior of this two-dimensional system using *XPPAUT* (even though it is easy to obtain a closed form solution). To analyze a differential equation using *XPPAUT* , you must create an input file that tells the program the names of the variables, parameters, and the equations. By convention, these files have the file extension `ode` and I will call them ODE files. Here is an ODE file for (2.1):

```
# linear2d.ode
#
# right hand sides
x'=a*x+b*y
y'=c*x+d*y
#
# parameters
par a=0,b=1,c=-1,d=0
#
# some initial conditions
init x=1,y=0
#
# we are done
done
```

I have included some comments denoted by lines starting with #; these are not necessary but can make the file easier to understand. The rest of the file is fairly straightforward (I hope). The values given to the parameters are optional; by default they are set to zero. The `init` statement is also optional. The minimal file for this system has 4 lines:

```
x'=a*x+b*y
y'=c*x+d*y
par a,b,c,d
done
```

Of course in this minimal file, all parameters are set to zero as are the initial conditions. Use a text editor to type in the first file exactly as it is shown or download the files. Name the file `linear2d.ode` and save it. That's it - you have written an ODE file. I give a number of practice examples at the end of this section. The minimal steps are:

- Use an editor to open up a text file.
- Write the differential equations in the file; one per line.
- Use the `par` statement to declare all the parameters in your system. Optionally define initial conditions with the `init` statement.
- End the file with the statement `done`
- Save and close the file.

NOTE The equation reader is case-insensitive so that `AbC` and `abC` are treated as identical.

WARNING In statements declaring initial conditions and parameters, **do not ever** put spaces between the variable and the "=" sign and the number. *XPPAUT* uses spaces as a delimiter. Always write `a=2.5` and **never** write `a = 2.5`.

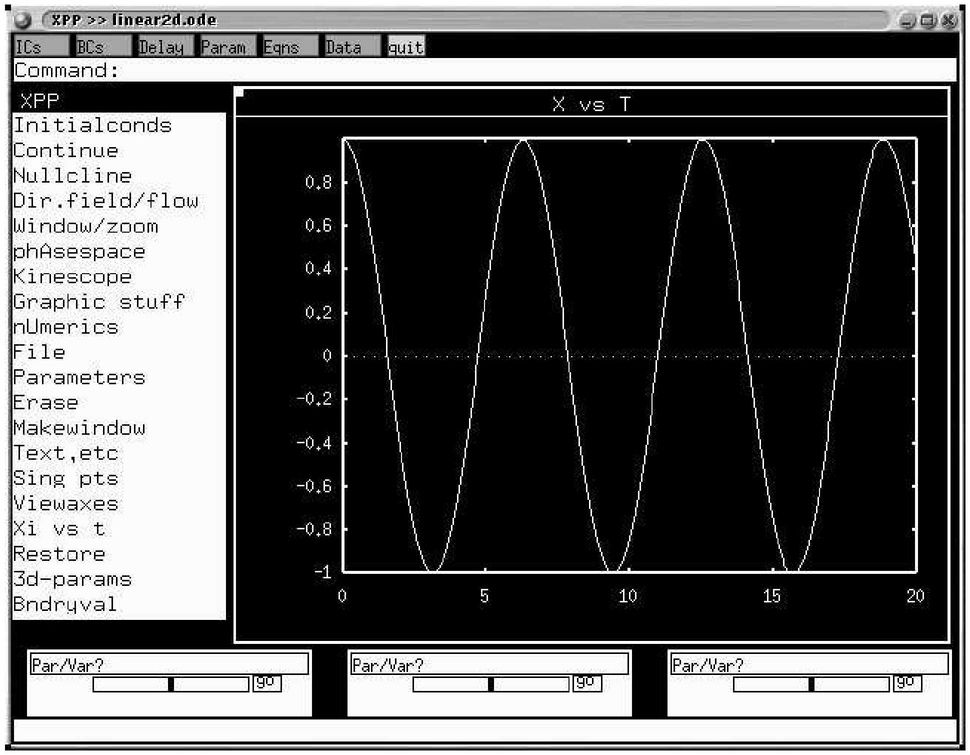


Figure 2.1. *The main XPPAUT window*

2.2 Running the program

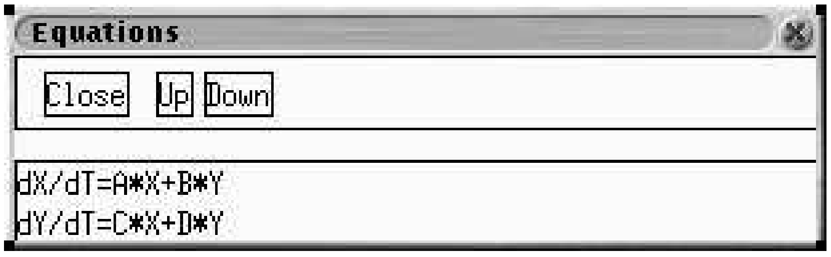
Run *XPPAUT* by typing

```
xpp linear2d.ode
```

Replace `xpp` with whatever you have decided to call the executable with all the desired command line options. (*If you are using winpp, click on the* **winpp** *icon; then choose the file from the file selection dialog box.*) A single window will appear unless you start *XPPAUT* with all the windows visible (`xppaut -allwin`, see Chapt 1).

2.2.1 The main window

The **Main Window** contains a large region for graphics, menus, and various other gadgets. It is illustrated in Figure 2.1. Commands are given either by clicking on the menu items in the left column with the mouse or tapping keyboard shortcuts. After a while, as you become used to *XPPAUT* , you will use the keyboard shortcuts more often. I will tell you the full commands and the keyboard shortcuts. In general, the



```
dX/dT=A*X+B*Y
dY/dT=C*X+D*Y
```

Figure 2.2. *The Equation Window.*

keyboard shortcut is the first letter of the command unless there is ambiguity (such as `Nullcline` and `nUmerics`) and then, it is just the capitalized letter (**N** and **U** respectively). Unlike Windows keyboard shortcuts, the letter key alone is sufficient and it is not necessary to press the Alt/Ctrl key at the same time. The top region of the **Main Window** is for typed input such as parameter values. The bottom of the **Main Window** displays information about various things as well as a short description of the highlighted menu item. The three little boxes with the words `parameter` are sliders to let you change parameters and initial data. Across the top of the **Main Window** you will see a menu bar of turquoise buttons. Clicking on these opens up a variety of windows. In addition, you may or may not have several other yellow buttons which are shortcuts to commands. These are defined by the user in the ODE file as options (see Appendix B) or in the *XPPAUT* resource file (see Chapt 1).

Click on the button labeled `Eqns` and the **Equation Window** will appear as shown in figure 2.2. This allows you to see the differential equations that you are solving. We will describe the other windows as the tutorial progresses. Clicking on the `Close` button closes this window.

2.2.2 Quitting the program

To exit *XPPAUT* , click `File Quit Yes` (**F Q Y** .)

2.3 Solving the equation, graphing, and plotting.

Here, we will solve the ODE, use the mouse to select different initial conditions, save plots of various types, and create files for printing.

Computing the solution. In the **Main Window** , you should see a box with axes numbers. The title in the window should say `X vs T` which tells you that the variable `X` is along the vertical axis and `T` along the horizontal. The plotting range is from 0 to 20 along the horizontal and -1 to 1 along the vertical axis. When a solution is computed, this view will be shown. Click on `Init Conds Go` (**I G**) in the **Main Window**. A solution will be drawn followed by a beep. As one would expect given the differential equations, the solution looks like a few cycles of a cosine wave.

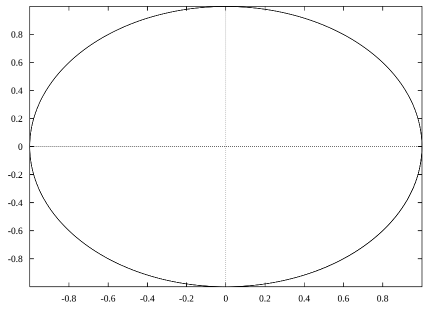


Figure 2.3. *Phase plane for the linear 2d problem.*

Changing the view. To plot `Y` versus `T` instead of `X`, just click on the command `Xi vs t` **X**) and choose `Y` by backspacing over `X`, typing in `Y` and typing **Enter** .

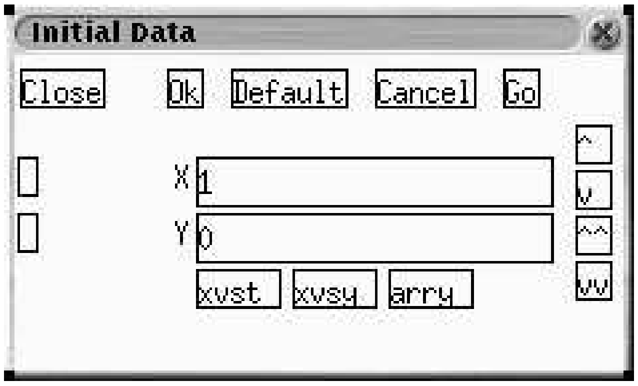
Many times, you may want to plot a phase-plane instead, that is `X` vs `Y`. To do this, click on `Viewaxes 2D` (**V 2**) and a dialog box will appear. Fill it in as follows:

`X-axis:` X	`Xmax:` 1
`Y-axis:` Y	`Ymax:` 1
`Xmin:` -1	`Xlabel:`
`Ymin:` -1	`Ylabel:`

Click on `OK` when you are done. (Note that you could have filled in the labels if you had wanted, but for now, there is no reason to.) You should see a nice elliptical orbit in the window. This is the solution in the phaseplane (cf Fig. 2.3).

Shortcuts. Click on the button `ICs` in the **Main Window** to bring up the **Initial Data Window**. This window shows the current initial data. There is a very simple way to view the phaseplane or view variables versus time. Look at the **Initial Data Window** (Fig 2.4). You will see that there are little boxes next to the variable names. Check the two boxes next to `X` and `Y`. Then at the bottom of the **Initial Data Window**, click the `XvsY` button. This will plot a phaseplane and automatically fit the window to contain the entire trajectory. This is a shortcut and does not give you the control that the menu command does. (For example, the window is always fit to the trajectory, and no labels are added or changed. Nor can you plot auxiliary quantities with this shortcut.) To view one or more variables against time, just check the variables you want to plot (up to 10) and click on the `XvsT` button in the **Initial Data Window**.

You should have a phaseplane picture in the window. (If not, get one following the above instructions or using the shortcut.) Click on `Init Conds Mouse` (**I M** .) Use the mouse to click somewhere in the window. You should see a new trajectory drawn. This, too, is an ellipse. Repeat this again to draw another trajectory. If you get tired of repeating this, try `Init Conds mIce` (**I I**) which, being "mice"



X 1
Y 0

Figure 2.4. *The initial conditions window.*

is many mouses. Keep clicking in the window. When you are bored with this, click either outside the window or tap the escape key, **Esc** .

Click on `Erase` and then `Restore` (**E** **R**). Note that all the trajectories are gone except the latest one. *XPPAUT* only stores the latest one. There is a way to store many of them, but we will not explore that for now.

Printing the picture.

XPPAUT does not directly send a picture to your printer. Rather, it creates a postscript file which you can send to your printer. If you don't have postscript capabilities, then you probably will have to use the alternate method of getting hardcopy. (Note that Word supports the import of PostScript and Encapsulated PostScript, but can only print such pictures to a postscript printer. You can download a rather large program for Windows called GhostView which enables you to view and print postscript on non-postscript printers. Linux and other UNIX distributions usually have a PostScript viewer included.)

Here is how to make a PostScript file. Click on `Graphics Postscript` (**G** **P**) and a dialog box will appear asking for three things: (i) Black and White or Color (ii) Landscape or Portrait; (iii) and the Fontsize for the axes. Accept all the defaults for now by just clicking `Ok` . You will then be asked for a filename. The File Selector box is shown in the figure 2.5. You can: move up or down directory trees by clicking on the `<>`; choose files by clicking on them; scroll up or down by clicking on the up/down arrows on the left or using the arrow keys and the PageUp/PageDown keys on the keyboard; change the wild card; or type in a filename. For now, you can just click on `Ok` and a PostScript plot will be created and saved. The file will be called `linear2d.ode.ps` but you can call it anything you want.

Once you have the postscript file, you can type

```
lpr filename
```

on UNIX. In Windows, if your computer is hooked up to a postscript printer, then this usually does the trick:

```
copy filename lpt1:
```

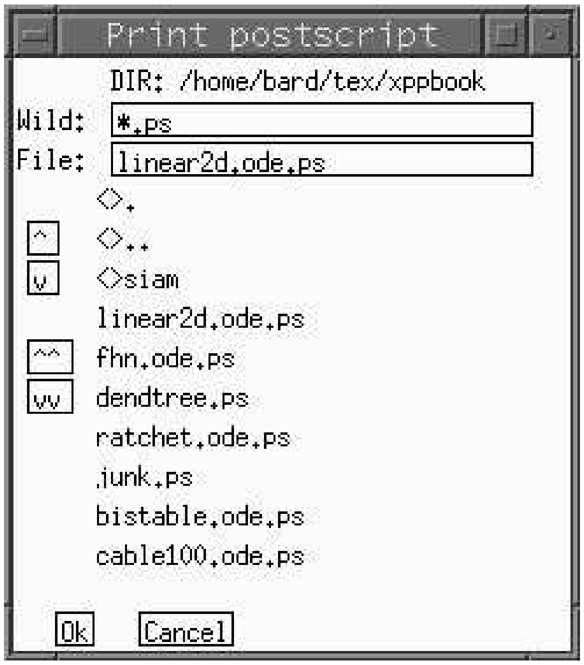


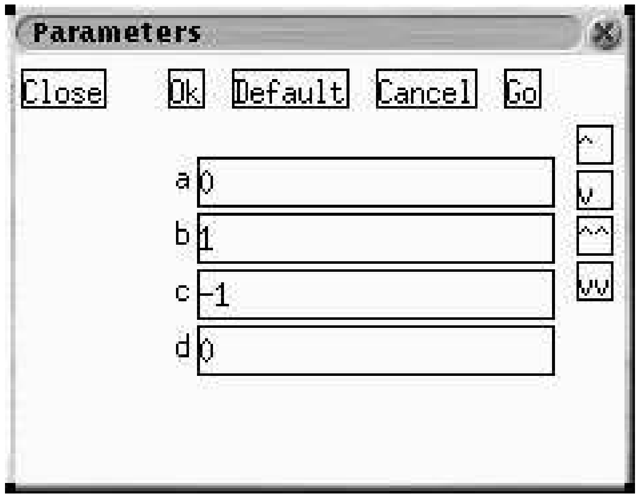
Figure 2.5. *File selector.*

In Windows, if you don't have a postscript printer, it is still easy to print the postscript files to your printer. However, you have to download a program called GhostView. This allows you to print postscript onto non-postscript printers. Most Windows distributions of GhostView come with a command-line utility called `gsprint.exe` in the `\Ghostgum\gsview` directory. Either of the following two lines work for me:

```
gsprint myfile.ps
gsprint -colour myfile.ps
```

where I use the latter for color printouts.

Other ways to get hardcopy. Another way to get hardcopy which you can import into documents is to grab the image from the screen. In Windows, click on `Alt+PrtSc` after making the desired window active. For some servers such as the MiX server, this will grab the entire X desktop window. Paste this into the Paint accessory and then use the tools in Paint to cut out what you want. Alternatively, you can download a number of programs that let you capture areas of the screen. In the UNIX environment, you can capture a window using `xv`, an excellent utility that is free and available for most UNIX versions. All of the screenshots in this tutorial were captured with `xv`. Finally, you can capture the screen (or a series of screen images) with the `Kinescope Capture` command and then write these to disk with the `Kinescope Save` command. This produces a gif file that is usable



a 0
b 1
c -1
d 0

Figure 2.6. *The parameter window.*

by many software packages including most browsers.

Getting a good window. If you have computed a solution and don't have a clue about the bounds of the graph, let *XPPAUT* do all the work. Click on `Window/zoom (F)it` and the window will be resized to a perfect fit. The shortcut is **W F** and you will likely use it a lot!

2.4 Changing Parameters and Initial Data.

There are many ways to vary the parameters and initial conditions in *XPPAUT*. We have already seen how to change the initial data using the mouse. This method works for any n-dimensional system as long as the current view is a phaseplane of two variables. Here are two other ways to change the initial data:

- From the main menu click on `Init Conds New` and manually put them in at the prompts. You will be prompted for each variable in order. (For systems with hundreds of variables, this is not a very good way to change the data!) However, if you get tired, just tap `Esc` and the remaining variables will assume their current initial conditions.

- In the **Initial Data Window** , you can edit the particular variable you want to change. Just click in the window next to the variable and edit the value. Then click on the `Go` button in the **Initial Data Window**. If there are many variables, you can use the little scroll buttons on the right to go up and down a line or page at a time. If you click the mouse in the text entry region for a variable, you can use the **PageUp** etc keys to move around. Clicking **Enter** rolls around in the displayed list of initial conditions. The `Default` button returns the initial data to those with with the program started. If you don't want to run the simulation, but have set the initial data, *you must* click on the `Ok` button in the **Initial Data Window** for the new initial data to be recognized.

- Attach the variable to a slider. Sliders are described next.

There are many ways to change parameters as well. Here are three of them:

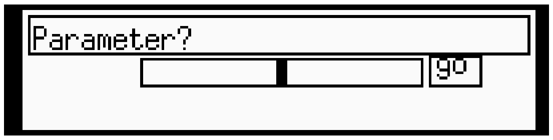
- From the **Main Window** , click on `Parameters` . In the command line of the **Main Window** , you will be prompted for a parameter name. Type in the name of a parameter that you want to change. Click on `Enter` to change the value and `Enter` again change another parameter. Click on `Enter` a few times to get rid of the prompt.
- Bring up the **Parameter Window** by clicking on turquoise `Param` button at the top of the **Main Window**. In the **Parameter Window** (shown in figure 2.6), type in values next to the parameter you want to change. Use the scroll buttons or the keyboard to scroll around. As in the **Initial Data Window** , there are four buttons across the top. Click on `Go` to keep the values and run the simulation; click on `Ok` to keep the parameters without running the simulation. Click on `Cancel` to return to the values since you last pressed `Go` or `Ok` . The `Default` button returns the parameters to the values when you started the program.
- Use the little sliders (Fig 2.7). We will attach the parameter `d` to one of the sliders. Click on one of the unused parameter sliders. Fill in the dialog box as follows:

`Parameter:` d
`Value:` 0
`Low:` -1
`High:` 1

and click `Ok` . You have assigned the parameter `d` to one of the sliders and allowed it to range between -1 and 1. Grab the little slider with the mouse and move it around. Watch how `d` changes. Now click on the tiny `go` button in the slider. The equations will be integrated. Move the slider some more and click on the `go` button to get another solution. The sliders can also be attached to initial data. Just choose a variable instead of a parameter.

2.5 Looking at the numbers - the Data Viewer.

In addition to the graphs that *XPPAUT* produces, it also gives you access to the actual numerical values from the simulation. You can bring up the **Data Viewer** by clicking on the turquoise button labeled `Data` at the top of the **Main Window**. The **Data Viewer** shown in Figure 2.8 has many buttons, some of which we will use later in the book. The main use of this is to look at the actual numbers



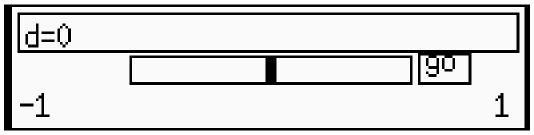
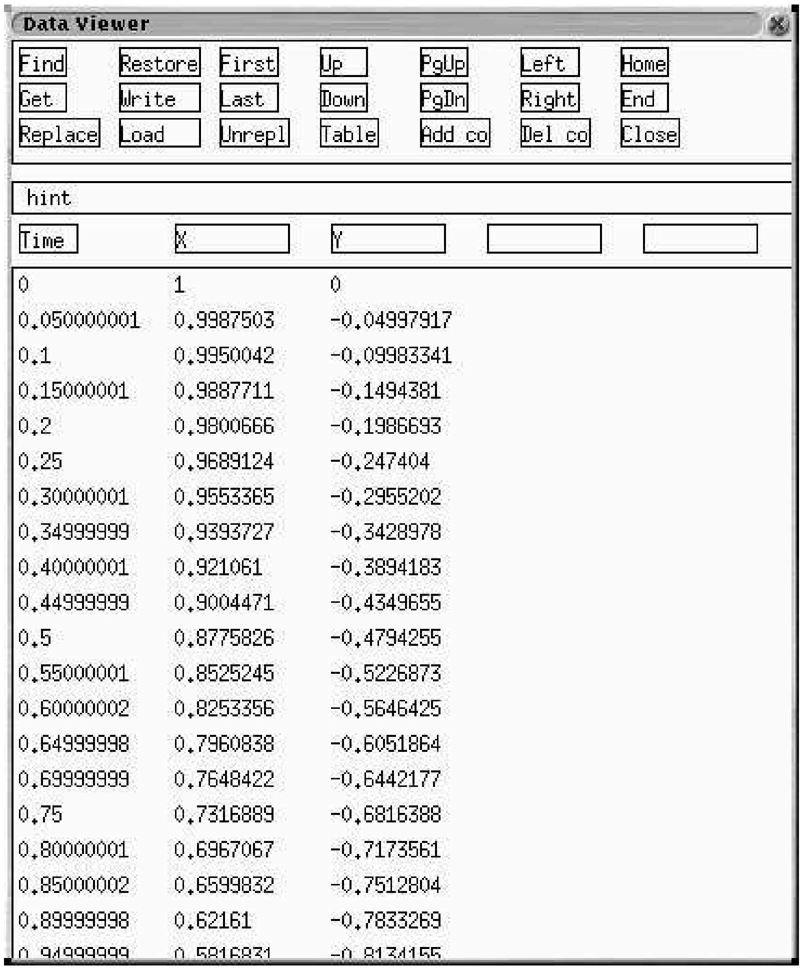


Figure 2.7. *Top: Unused parameter slider. Bottom: Used parameter slider.*



```
Time          X             Y
0             1             0
0.050000001   0.9987503     -0.04997917
0.1           0.9950042     -0.09983341
0.15000001    0.9887711     -0.1494381
0.2           0.9800666     -0.1986693
0.25          0.9689124     -0.247404
0.30000001    0.9553365     -0.2955202
0.34999999    0.9393727     -0.3428978
0.40000001    0.921061      -0.3894183
0.44999999    0.9004471     -0.4349655
0.5           0.8775826     -0.4794255
0.55000001    0.8525245     -0.5226873
0.60000002    0.8253356     -0.5646425
0.64999998    0.7960838     -0.6051864
0.69999999    0.7648422     -0.6442177
0.75          0.7316889     -0.6816388
0.80000001    0.6967067     -0.7173561
0.85000002    0.6599832     -0.7512804
0.89999998    0.62161       -0.7833269
0.94999999    0.5816831     -0.8134155
```

Figure 2.8. *The Data Viewer.*

from a simulation. The independent variable occupies the left-most column and the dependent variables filling in the remaining windows. Click on the top of the **Data Viewer**to make it the active window. The arrow keys and the **PageUp** , **PageDown** , **Home** , and **End** keys (as well as their corresponding buttons) do all the obvious things. Left and right keys scroll horizontally - a useful feature if you have many variables. I mention three buttons of use:

`Find` brings up a dialog box prompting you for the name of a column and a value. If you click on `Ok` , *XPPAUT* will find the entry that is closest and bring that row to the top. You can find the maximum and minimum, for example, of a variable by choosing a really big or small number in the `Find` dialog.

`Get` loads the top line of the **Data Viewer** as initial data.

`Write` writes the entire contents of the browser to a text file that you specify.

2.6 Saving and restoring the state of *XPPAUT*.

Often you will have a view, a set of parameters, and initial data that you want to keep. You can save the current state of *XPPAUT* by clicking on `File Write set` (**F W**) in the **Main Window**. This bring up a file selection box. Type in a filename – the default extension is `.set`. The resulting file is an ASCII file that is human and computer readable. The first and last few lines look like:

```
## Set file for linear2d.ode on Fri Aug  4 13:53:31 2000
2   Number of equations and auxiliaries
4   Number of parameters
# Numerical stuff
1    nout
40    nullcline mesh

.......

RHS etc ...
dX/dT=A*X+B*Y
dY/dT=C*X+D*Y
```

Once you quit *XPPAUT* , you can start it up again and then use the `File Read set` to load up the parameters etc that you saved.

Now you should quit the program. We will look at a nonlinear equation next, find fixed points, and draw some nullclines and direction fields. To quit click on `File Quit Yes` (**F Q Y**).

2.6.1 Command summary

`Initialconds Go` computes a trajectory with the initial conditions specified in the **Initial Data Window** . (**I G**)

`Initialconds Mouse` computes a trajectory with the initial conditions specified by the mouse. `Initialconds m(I)ce` lets you specify many initial conditions. (**I M** or **I I**)

`Erase` erases the screen.(**E**)

`Restore` redraws the screen. (**R**)

`Viewaxes 2D` lets you define a new 2D view. (**V 2**)

`Graphic stuff Postscript` allows you to create a postscript file of the current graphics. (**G P**)

`Kinescope Capture` allows you to capture the current view into memory and `Kinescope Save` writes this to disk.

`Window/zoom (F)it` fits the window to include the entire solution. (**W F**).

`File Quit` exits the program. (**F Q**)

`File Write set` saves the state of *XPPAUT*. (**F R**).

`File Read set` restores the state of *XPPAUT* from a saved `.set` file. (**F R**).

2.7 A nonlinear equation.

Here we want to solve a nonlinear equation. We will choose a planar system since there are many nice tools available for analyzing two-dimensional systems. A classic model is the Fitzhugh-Nagumo equation which is used as a model for nerve conduction. The equations are:

$$\frac{dV}{dt} = I + V(1-V)(V-a) - w \qquad (2.2)$$
$$\frac{dw}{dt} = \epsilon(V - \gamma w)$$

with parameters I, a, ϵ, γ. Typical values are $a = .1, I = 0, \epsilon = .1$, and $\gamma = 0.25$. Let's write an ODE file for this:

```
# Fitzhugh-Nagumo equations
v'=I+v*(1-v)*(v-a) -w
w'=eps*(v-gamma*w)
par I=0,a=.1,eps=.1,gamma=.25
@ xp=V,yp=w,xlo=-.25,xhi=1.25,ylo=-.5,yhi=1,total=100
@ maxstor=10000
done
```

We have already seen the first four lines: (i) lines beginning with a # are comments, (ii) the next two lines define the differential equations and (iii) the line beginning with `par` defines the parameters and their default values. The penultimate line beginning with the `@` sign is a directive to set some of the options in *XPPAUT*. These could all be done within the program, but this way everything is all set up for you. Details of these options are found in the appendix. For the curious, these options set the x-axis (`xp`) to be the `V` variable, the y-axis (`yp`) to be the `w` variable, the plot range to be $[-.25, 1.25] \times [-.5, 1]$, and the total amount of integration time to be 100. The last option `@ maxstor=10000` is a very useful one. *XPPAUT* allocates enough storage to keep 4000 time points. You can make it allocate as much as you want with this option. Here I have told *XPPAUT* to allocate storage for 10000 points. Type this in (or download it; all of the files are available for downloading at the author's web page) and save it as `fhn.ode`.

2.7.1 Direction fields

Run this by typing `xpp fhn.ode`. The usual windows will pop up. One of the standard ways to analyze differential equations in the plane is to sketch the *direction fields.* Suppose that the differential equation is:

$$x' = f(x,y) \qquad y' = g(x,y).$$

The phaseplane is divided into a grid and at each point in the grid, (x,y), a vector is drawn with (x,y) as the base and $(x+sf(x,y), y+sg(x,y))$ as the terminal point where s is a scaling factor. This so-called direction field gives you a hint about how trajectories move around in the plane. *XPPAUT* lets you quickly draw the direction field of a system. Click on `Dir.field/flow (D)irect Field` (**D D**) and then accept the default of 10 for the grid size by clicking `Enter` . A bunch of vectors will be drawn on the screen, mainly horizontal. They are horizontal because ϵ is small so that there is little change in the w variable. The length of the vectors is proportional to the magnitude of the flow at each point. At the head of each vector is a little bead. If you want to have scaled direction fields which don't take into account the magnitude of the vector field, just click on `Dir.field (S)caled Dir. Fld` (**D S**) and use the default grid size. (I prefer pure direction fields, but this is a matter of taste.) Click on `Initialconds m(I)ce` and to experiment with a bunch of different trajectories. Note how the vectors from the direction field are tangent to the trajectories. See figure 2.9.

2.7.2 Nullclines and fixed points

A powerful technique for the analysis of planar differential equations and related to the direction fields is the use of *nullclines.* Nullclines are curves in the plane along which the rate of change of one or the other variable is zero. The x-nullcline is the curve where $dx/dt = 0$, that is, $f(x,y) = 0$. Similarly the $y-$nullcline is the curve where $g(x,y) = 0$. The usefulness of these curves is that they break the plane up into regions along which the derivatives of each variable have a constant sign. Thus,

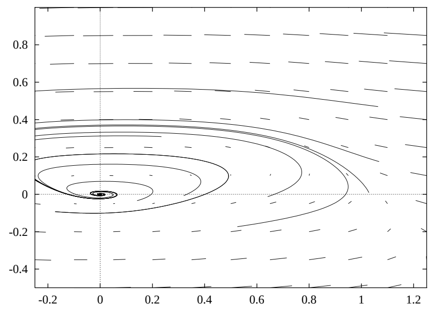


Figure 2.9. *Direction fields and some trajectories for the Fitzhugh-Nagumo equations.*

the general direction of the flow is easy to determine. Furthermore, any place that they intersect represents a fixed point of the differential equation.

XPPAUT can compute the nullclines for planar systems. To do this, just click on `Nullcline New` (**N N**). You should see two curves appear; a red one representing the V-nullcline and a green one representing the W-nullcline. The green one is a straight line and the red is a cubic. They intersect just once: there is a single fixed point. Move the mouse into the phaseplane area and hold it down as you move it. At the bottom of the **Main Window** you will see the x and y coordinates of the mouse. The intersection of the nullclines appears to be at (0,0). Figure 2.10 shows a printout along with a representative trajectory when the parameter $I = 0.3$.

The stability of fixed points is determined by linearizing about them and finding the eigenvalues of the resulting linear matrix. *XPPAUT* will do this for you quite easily. *XPPAUT* uses Newton's method to find the fixed points and then numerically linearizes about them to determine stability. To use Newton's method, a decent guess needs to be provided. For planar systems, this is easy to do – it is just the intersection of the nullclines. In *XPPAUT* fixed points and their stability are found using the `Sing pts` command as "singular points" is a term sometimes used for fixed points or equilibrium points. Click on `Sing pts Mouse` (**S M**) and move the mouse to near the intersection of the nullclines. Click the button and a message box will appear on the screen. Click on `No` since we don't need the eigenvalues. A new window will appear that contains information about the fixed points. The stability is shown at the top of the window. The nature of the eigenvalues follows:**c+** denotes the number of complex eigenvalues with positive real part; **c-** is the number of complex eigenvalues with negative real part; **im** is the number of purely imaginary eigenvalues; **r+** is the number of positive real eigenvalues; and **r-** is the number of negative real eigenvalues. Recall that a fixed point is linearly stable if all of the eigenvalues have negative real parts. Finally, the value of the fixed points is shown under the line. As can be seen from this example, there are two complex eigenvalues with negative real parts: the fixed point is (0,0). (*XPPAUT* reports a very small nonzero fixed point due to numerical

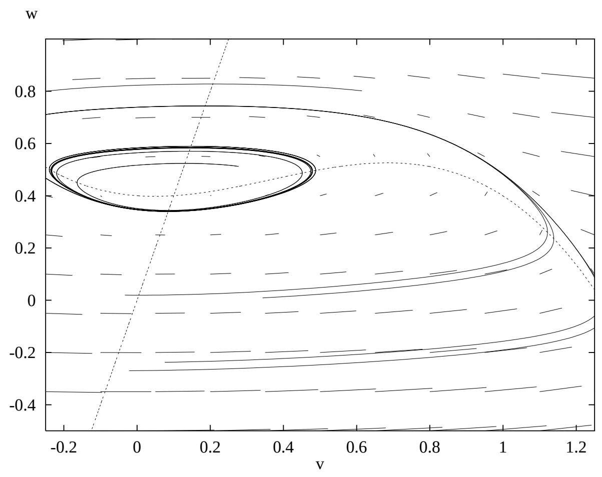


Figure 2.10. *Nullclines,direction fields, trajectories for* $I = 0.4$ *in the Fitzhugh-Nagumo equations.*

error.) Integrate the system using the mouse, starting with initial conditions near the fixed point. (In the **Main Window** , tap **I** **I** .) Note how solutions spiral into the origin as is expected when there are complex eigenvalues with negative real parts.

For nonplanar systems of differential equations, you must provide a direct guess. Bring up the **Initial Data Window**, type in your guess, and click on `Ok` in the **Initial Data Window**. Then from the **Main Window** , click on `Sing Pts Go` (**S** ,`G`).

Another way to get fixed points is to use the `Sing Pts monte(C)ar` command, which randomly picks starting guesses in a range and then tries to find them with Newton's method. You have to provide the ranges for starting guesses as well as the number of guesses. The fixed points found are listed in the Data Viewer. You should use this method only if you have no clue where the fixed points lie.

Bring up the **Parameter Window** . Change the parameter `I` from 0 to 0.4 in the **Parameter Window** and click on `Ok` in the **Parameter Window** . In the **Main Window** , erase the screen and redraw the nullclines: `Erase Nullclines New` (**E** **N** **N**). The fixed point has moved up. Check its stability using the mouse (`Sing pts Mouse`). The fixed point should be (0.1,0.4). Use the mouse to choose a bunch of initial conditions in the plane. All solutions go to a nice limit cycle. That is, they converge to a closed curve in the plane representing a stable periodic solution.

Let's make a nice picture that has the nullclines, the direction fields and a few representative trajectories. Since *XPPAUT* only keeps the last trajectory computed, we will "freeze" the solutions we compute. You can freeze trajectories automatically or one at a time. We will do the former. Click on `Graphic stuff (F)reeze (O)n freeze` (**G** **F** **O**) to permanently save computed curves. Up to

26 can be saved in any window. Now use the mouse to compute a bunch of trajectories. Draw the direction fields by clicking `Dir.field/flow (D)irect Field` (**D D**). Finally, lets label the axes. Click on `Viewaxes 2D` (**V 2**) and the 2D view dialog will come up. Change nothing but the labels, (the last two entries) and put `V` as the **Xlabel** and `w` as the **Ylabel**. Click on `Ok` to close the dialog. Finally, since the axes are confusing in the already busy picture, click on `Graphic stuff aXes opts` (**G X**) and in the dialog box change the 1's in the entries **X-org(1=on)** and **Y-org(1=on)** to 0's to turn off the plotting of the X and Y axes. Click `Ok` when you are done. Now create a postscript file, `Graphic stuff (P)ostscript` (**G P**) and accept all the defaults. Name the file whatever you want and click on **Ok** in the file selection box. Figure 2.10 shows the version that I have made. Yours will be slightly different. If you want to play around some more; turn off the automatic freeze option : `Graphic stuff Freeze Off freeze` (**G F** O); and delete all the frozen curves: `Graphic stuff Freeze Remove all` (**G F R**).

2.7.3 Command Summary

`Nullcline New` draws nullclines for a planar system. (**N N**)

`Dir.field/flow (D)irect Field` draws direction fields for a planar system. (**D D**)

`Sing pts Mouse` computes fixed points for a system with initial guess specified by the mouse. (**S M**)

`Sing pts Go` computes fixed points for a system with initial guess specified by the current initial conditions. (**S G**)

`Graphic stuff Freeze On Freeze` will permanently keep computed trajectories in the current window. (**G F O**)

`Graphic stuff Freeze Off Freeze` will toggle off the above option. (**G F O**)

`Graphic stuff Freeze Remove all` delete all the permanently stored curves. (**G F R**)

`Graphic stuff aXes opts` lets you change the axes. (**G X**).

`Viewaxes 2D` allows you to change the 2D view of the current graphics window and to label the axes. (**V 2D**).

2.8 The most important numerical parameters.

XPPAUT has many numerical routines built into it and thus there are many numerical parameters that you can set. These will be dealt with in subsequent sections of the book where necessary. However, the most common things you will want to change are the total amount of time to integrate and the step size for integration. You may also want to change the method of integration from the default fixed step

Runge-Kutta algorithm. To alter the numerical parameters, click on `nUmerics` (**U**) which produces a new menu. This is a top level menu so you can change many things before going back to the main menu. To go back to the main menu, just click on the `[Esc]-exit` or tap **Esc** . There are many entries on the numerics menu. The following four are the most commonly used:

`Total` sets the total amount of time to integrate the equations. (Shortcut: **T**)

`Dt` sets the size of the timestep for the fixed step size integration methods and sets the output times for the adaptive integrators. (Shortcut: **D**)

`Nout` sets the number of steps to take before *plotting* an output point. Thus, to plot every fourth point, change `Nout` to 4. For the variable step size integrators, this should be set to 1.

`Method` sets the integration method. There are currently 15 available. (Shortcut: **M**). They are described in the appendix.

When you are done setting the numerical parameters, just click on `Esc-exit` or tap the `Esc` key.

Exercises

2.1. Write a differential equation file for the damped pendulum:

$$\ddot{x} + a\dot{x} + \sin x = 0.$$

Hint: Rewrite this as a system of two first order equations:

$$x' = v \qquad v' = -av - \sin x.$$

Look at the phaseplane (plot x along the horizontal and v along the vertical axis) for this in the case of no damping ($a = 0$) and small positive damping. A good window to use is $[-8, 8] \times [-3, 3]$. Draw the direction field and classify all of the fixed points with and without damping. (In any message boxes related to the computation of fixed points, answer `No` in every case. If, by mistake, you click on `Yes`, then repeatedly tap the **Esc** key.) Draw some representative trajectories with and without damping. Plot your work in the damped and undamped case as a Postscript file.

2.2. Fire up the Fitzhugh-Nagumo equation. Increase the parameter I from zero and determine the value at which the fixed point becomes unstable. Continue to increase I and figure out the value at which the fixed point again becomes stable. Next, set $I = 0$ and change γ to 10, erase the screen and recompute the nullclines. How many fixed points are there? Determine their stability. Choose a bunch of initial conditions and see if you an detect a difference between the long-time behavior of the trajectory.

2.3. Write a differential equations file for the Lorenz equations

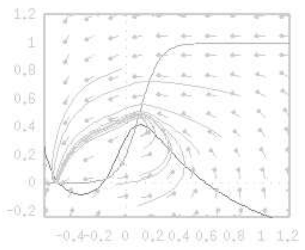
$$\begin{aligned} x' &= s(-x+y) \\ y' &= rx - y - xz \\ z' &= -bz + xy \end{aligned}$$

with initial conditions $x = -7.5, y = -3.6, z = 30$, and starting parameter values of $r = 27, s = 10, b = 2.66$. (Don't forget to put the multiplication sign `*` wherever there are products!) Run *XPPAUT* and plot x against time. Then plot x and z together. (Hint click on the variables x and y in the **Initial Data Window** and then click the `xvsy` button in the **Initial Data Window** .) You might want to go into the numerics menu and make `Dt` smaller and the total amount of time longer - *e.g.* set `Dt=.025` and `Total=40`.

2.4. Sketch phaseplanes for the following systems in the window, $[-3,3] \times [-3,3]$

1. $x' = y - 2x$, $y' = x + y$;
2. $x' = x$, $y' = x^2 + y^2 - 1$;
3. $x' = x(1 - x/2 - y)$, $y' = y(x - 1 - y/2)$
4. $x' = y^2$, $y' = x$;
5. $x' = 2 - x - y^2$, $-y(x^2 + y^2 - 3x + 1)$.

(Hint: Only make one ODE file. Within *XPPAUT* , click on `File Edit RHS's` to edit the right-hand sides. Click on `OK` when you are done. Make sure you include the `*` for multiplication!)



Chapter 3

Writing ODE files for differential equations

3.1 Introduction

XPPAUT is equipped with a powerful parser that has many useful functions and tools that make it easy to solve problems that are related to dynamical systems. Furthermore there are many tricks that you can use to solve very complicated problems. In the last chapter of this book, I will provide a survey of the more obscure tricks. Perhaps the best way to describe the range of ODE files is to give you a number of examples sorted by problem type. Appendix D gives a complete description of the language for the creation of ODE files. Every ODE file is a plain text file. The total length of lines must be less that 1000 characters. Line continuation is done with the \ character:

```
x' = y \
 + z + \
w
```

This is interpreted as `x' = y+z+w`.

Every ODE file consists of declarations of the equations that you want to solve, the parameters involved, and any user defined functions, etc that you will need. Every ODE file must end with the `done` statement. Other than that, the form of the files is pretty flexible. The name of anything that is defined in *XPPAUT* must be fewer than 10 characters. The allowable characters are the letters of the alphabet, numbers, the underscore symbol "_". Other symbols are not recommended since they may be confused with symbols that have meaning to *XPPAUT*. Do not start names with a number.

Most of the input that you type into an ODE file is pretty much as you'd expect and similar to how you would write it on paper. The main exceptions to this are (i) you must use the `*` symbol between items that are to be multiplied and (ii) *XPPAUT* is case insensitive.

All of the examples here and in the rest of the book are available online so that you need not type them in. I also encourage you to run the example files and put in some initial conditions or parameters so that you get used to simulation. In many of the examples, I suggest some interesting things to explore.

3.2 Ordinary differential equations and maps.

The easiest types of equations to type in are ordinary differential equations (ODE's) or maps. You type these in almost exactly in the way you would write them on paper. There are two different ways to declare a differential equation. Consider the ODE:

$$\frac{dx}{dt} = -x.$$

Then the corresponding line in an ODE file will be either

```
x' = -x
```

or

```
dx/dt = -x
```

I will generally use the former since it is less typing. (The spaces are not required here and are just for readability.)

There are two different ways to define a map. Consider

$$x_{n+1} = x_n/2.$$

You could write this in two ways:

```
x' = x/2
```

or

```
x(t+1) = x/2
```

Both are interpreted the same way. Again, I will use the former as it is less typing.

Higher order differential equations and maps must be rewritten as systems of first order equations. Thus, the classic damped spring:

$$\frac{d^2x}{dt^2} + f\frac{dx}{dt} + k(x-l) = 0$$

would be written as

$$\frac{dx}{dt} = v \qquad \frac{dv}{dt} = -fv - k(x-l)$$

and the corresponding ODE file is written as

```
# the spring
x'=v
v'=-f*v-k*(x-l)
par f=0,l=1,k=1
done
```

where I have assigned some arbitrary values to the parameters as well as a comment.

Consider the delayed logistic map:

$$x_{n+1} = rx_n(1 - x_{n-1})$$

which is a second order map. To solve this in *XPPAUT* we rewrite it as a system of two first order maps:

$$x_{n+1} = y_n \qquad y_{n+1} = x_{n+2} = rx_{n+1}(1 - x_n) = ry_n(1 - x_n).$$

Thus the ODE file is

```
# delayed logistic
x(t+1)=y
y(t+1)=r*y*(1-x)
par r=2.1
init y=.25
@ total=200
done
```

I have added an initial condition statement for y. I have also set the number of iterations to 200 instead of the default of 20. The “t+1” tells *XPPAUT* that the file should be considered a model with discrete dynamics so that the method of solving the equations is automatically set to `discrete`.

I could also save typing by also writing this file as

```
# delayed logistic
x'=y
y'=r*y*(1-x)
par r=2.1
init y=.25
@ meth=discrete,total=200
done
```

(Here, since I don't use the construction “t+1”, I have to tell *XPPAUT* that the model is discrete. *XPPAUT* defaults to a continuous differential equation.) Start up *XPPAUT* with this file (*e.g.* `xpp delaylog.ode`*). Solve the equations by clicking* `Initialconds Go` *(***I G** *). Then expand the viewing window by clicking on* `Window Window` *and changing* `X Hi` *to 200. Change the parameter r making it smaller or bigger and watch what happens. If r is too big, show that solutions blow up (*i.e., they go out of bounds).

3.2.1 Nonautonomous systems

XPPAUT uses `t` to denote the independent variable. For differential equations and other continuous dynamical systems, `t` is continuous time. For maps, it is always a natural number $0, 1, 2, \ldots$. Consider the forced Duffing equation:

$$\frac{d^2x}{dt^2} + f\frac{dx}{dt} + ax(x^2 - 1) = c\cos\omega t.$$

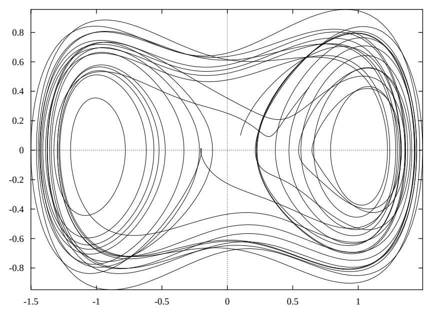


Figure 3.1. *Phase-plane of the forced Duffing equation*

As above, we write this as a pair of first order equations. The resulting ODE file, `duffing.ode` is

```
# forced duffing equation
x'=v
v'=a*x*(1-x^2) -f*v+c*cos(omega*t)
par a=1,f=.2,c=.3,omega=1
init x=.1,v=.1
done
```

Try running this equation. Change the total integration time to 200 by clicking on `nUmerics Total` and entering 200. Then click on `Esc` and integrate the equations with `Initialconds Go`. Click on `Window Fit` to fit the entire time series into the window. Create a new window clicking on `Makewindow Create`. Stretch it out to make it bigger. Then make a plot of the phase-plane by clicking the little boxes next to the variables `x` and `v` in the **Initial Data Window** and then clicking on the `xvsy` button in the **Initial Data Window**. You should see something like Figure 3.1.

In a similar vein, we can write nonautonomous maps in the same manner. For example the logistic equation with slowly increasing carrying capacity:

$$x_{n+1} = rx_n(1 - \frac{x_n}{1+an})$$

becomes

```
# time dependent carrying capacity
par a=.01,r=1.8
init x=.1
x(t+1)=r*x*(1-x/(1+a*t))
@ meth=discrete,total=200
done
```

`sin(x)`	$\sin(x)$	`cos(x)`	$\cos(x)$	`tan(x)`	$\tan(x)$
`atan2(x,y)`	$\tan^{-1}(y/x)$	`asin(x)`	$\sin^{-1}(x)$	`acos(x)`	$\cos^{-1}(x)$
`atan(x)`	$\tan^{-1}(x)$	`sinh(x)`	$\sinh(x)$	`cosh(x)`	$\cosh(x)$
`tanh(x)`	$\tanh(x)$	`x**y,x^y`	x^y	`exp(x)`	e^x
`abs(x)`	$\|x\|$	`ln(x)`	$\ln(x)$	`log(x)`	$\ln(x)$
`log10(x)`	$\log_{10}(x)$	`sqrt(x)`	$\sqrt{x}$	`max(x,y)`	$\max(x,y)$
`min(x,y)`	$\min(x,y)$	`sign(x)`	$x/\|x\|$	`heav(x)`	if $x \geq 0$ then 1 else 0
`flr(x)`	$\text{int}(x)$	`erf(x)`	$\text{erf}(x)$	`mod(x,y)`	x modulo y
`erfc(x)`	$1-\text{erf}(x)$	`x & y`	x AND y	`bessely(n,x)`	$Y_n(x)$
`x | y`	x OR y	`not(x)`	˜x	`besselj(n,x)`	$J_n(x)$

Table 3.1. *Names of standard XPPAUT functions*

3.3 Functions

XPPAUT comes with all of the usual functions you expect with the standard names. In addition there are a number of less standard functions that you can use in the program.

The following logical functions return 1 or 0 depending on the truth of the (in)equality, `x>y`, `x<y`, `x==y`, `x<=y`, `x>=y`, `x!=y`. The last means x NOT EQUAL y. (For all these logical operations, it is best to surround the right and left sides by parentheses.) Here are the remaining simple functions:

- `ran(x)` produces a uniformly distributed random number between 0 and x.
- `normal(x,s)` produces a normally distributed random number with mean x and standard deviation y.
- `if(x)then(y)else(z)` returns y if x is true, otherwise it returns z.
- `sum(x,y)of(z)` produces $\sum_{i'=x}^{i'=y} z$ where z is some expression involving the index `i'`. For example `sum(0,10)of(i')` evaluates to 55.

There are other complicated functions that act directly on variables; we will describe them shortly.

3.3.1 User defined functions

It is easy to define functions of up to 9 variables in *XPPAUT*. A function is defined with the statement such as:

```
f(x,y) = x/(x+y)
```

Functions can depend on other functions, but be careful of recursive definitions as *XPPAUT* does not check for this and will unceremoniously crash if such a function is called. (A legitimate recursive function will not cause a crash, e.g., one that stops such as `fac(x)=if(x>0)then(x*f(x-1))else(1)`.) For example, here is an ODE file, `wc.ode`, for the Wilson-Cowan equations:

```
# the wilson-cowan equations
u'=-u+f(a*u-b*v+p)
v'=-v+f(c*u-d*v+q)
f(u)=1/(1+exp(-u))
par a=16,b=12,c=16,d=5,p=-1,q=-4
@ xp=u,yp=v,xlo=-.125,ylo=-.125,xhi=1,yhi=1
done
```

Note that in the ODE file, I have defined the logistic function $f(x)$ and also included a line so that when the program is run, the graphics window displays the (u, v) phaseplane This is done on line beginning with the `@` symbol. Here, the statement `xp=u` says that the variable to be plotted on the x-axis is u, `yp=v` says to plot v on the y-axis, and the remaining statements set up the window size. Fire this up and look at the phaseplane as you change parameters. For example, draw the nullclines by clicking `Nullclines New` . Try choosing a bunch of different initial conditions with the `Initialconds Mice` command. Vary the parameter `a` by making it lower. Figure out at what point the limit cycle disappears. To automate this, attach the parameter `a` to one of the parameter sliders. Give it a range between 0 and 20.

3.4 Auxiliary and temporary quantities

In *XPPAUT* , the values of every one of the variables that define the dynamical system are available for plotting, storing, and manipulating. However, sometimes there are other quantities that you would like to see such as the total energy of a system or one of the currents in a model for a cell membrane. *XPPAUT* allows you to define these within an ODE file so that they are accessible for plotting, etc. This is done by writing a line that starts with `aux` followed by the definition of the quantity. E.g.

```
aux stuff=x+y*sin(t)
```

`stuff` will now be plottable. Let's take as an example the pendulum which satisfies:

$$ml^2\ddot{\theta} = -mgl\sin\theta$$

where l is the length, g is gravity, and m is the mass of the pendulum. θ is the angle of the pendulum with respect to the vertical axis. The kinetic energy is $K = m(l\dot{\theta})^2/2$ and the potential energy is $P = mgl(1 - \cos\theta)$. Thus, the total energy is

$$E = m(l\dot{\theta})^2/2 + mgl(1 - \cos\theta).$$

Let's write an ODE file for the pendulum and make the energy a plottable quantity. As usual, we want to first make the equation a system of two first order equations:

$$\dot{\theta} = v \qquad \dot{v} = -(g/l)\sin\theta.$$

The ODE file is

```
# the undamped pendulum
theta'=v
v'=-(g/l)*sin(theta)
par g=9.8,l=2,m=1
aux E=m*((l*v)^2/2+g*l*(1-cos(theta)))
done
```

Now if you run this, the energy will be accessible. You will find that it is constant along solutions to the differential equation, a hallmark of frictionless mechanical systems. In *XPPAUT* , auxiliary quantities that are made available for plotting etc are called **auxiliary variables**. Rewrite the above ODE file to incorporate a linear friction term with magnitude f:

$$\dot{\theta} = v \qquad \dot{v} = -(g/l)\sin\theta - fv.$$

Run this, plot the energy for $f = 0$ and $f = 0.1$, and notice how the energy dissipates to zero when there is friction.

3.4.1 Fixed variables

If you have a particularly complicated equation with quantities that are repeatedly used, then it is often useful for both computational and readability issues to define them as temporary quantities. Say the quantity is called `z`, then you just include a statement:

```
z=x+y*sin(t)
```

and that quantity is defined and can be used in the ODE . For example, consider the nonlinear oscillator:

$$\frac{dx}{dt} = x(a-R) - y(1+qR) \qquad \frac{dy}{dt} = y(a-R) + x(1+qR)$$

where $R = x^2 + y^2$. We should define R in the ODE file and then write the equations:

```
# standard nonlinear oscillator
R=x^2+y^2
x'=x*(a-R)-y*(1+q*R)
y'=y*(a-R)+x*(1+q*R)
par a=1,q=1
done
```

You can have many such temporary quantities and they can depend on each other. In *XPPAUT* these are called **fixed variables**.

IMPORTANT NOTES

- Auxiliary variables are not known to *XPPAUT* and so you cannot use them in any formulas.

- Fixed variables are not accessible to the user - they cannot be plotted, but can be used in formulas.
- The order of definition for fixed variables is important as they are evaluated in the order in which they are defined. Thus, if you define a quantity `x` that depends on another quantity `y`, then you should define `y` first or you will get unintended results.

Just remember if you want to use it in a formula, make it a fixed variable and if it is just an auxiliary quantity you want to plot, make it an auxiliary variable. You can use fixed variables in definitions of auxiliary variables but not vice versa.

In the oscillator example above, we could make the fixed variable `R` available for plotting by adding a statement:

```
aux myr=R
```

so that `myr` is the user accessible version of `R`.

3.4.2 Exercises

1. Write an ODE file for the Fibonacci recurrence:

$$f_{n+1} = f_n + f_{n-1} \qquad f_0 = f_1 = 1.$$

2. Write an ODE file for the Lotka-Volterra equations

$$\frac{dx}{dt} = ax - bxy \qquad \frac{dy}{dt} = -cy + dxy$$

with positive parameters a, b, c, d. Track the quantity:

$$Q = a \ln|y| + c \ln|x| - by - dx$$

with initial data, $x = y = \frac{1}{2}$.

3. Write a differential equation file for the Morris-Lecar equations defined as follows:

$$\begin{aligned} C\frac{dV}{dt} &= -g_L(V - E_L) - g_{Ca}m_\infty(V)(V - E_{Ca}) - g_K w(V - E_K) + I \\ \frac{dw}{dt} &= \phi\frac{w_\infty(V) - w}{\tau_w(V)} \\ m_\infty(V) &= 1/(1 + \exp(-(V - V_1)/V_2)) \\ w_\infty(V) &= 1/(1 + \exp(-(V - V_3)/V_4)) \\ \tau_w(V) &= 1/\cosh((V - V_3)/(2V_4)) \end{aligned}$$

where $\phi = 0.8$, $g_K = 8$, $g_{Ca} = 4.4$, $g_L = 2$, $C = 1$, $E_K = -84$, $E_{Ca} = 120$, $E_L = -60$, $V_1 = -1.2$, $V_2 = 9$, $V_3 = 2$, $V_4 = 15$, and $I = 90$. Integrate them and analyze them along the lines that you did in chapter 2 for the Fitzhugh-Nagumo equations. Look in the (V, w) phase plane and show that there is a coexistent periodic solution and a stable fixed point. (If the ODE file is too hard for you to figure out – then as a last resort, download the ODE file, `mlex.ode`.)

4. Write the third-order system:

$$x''' + ax'' + bx' + cx' = x^2$$

as a system of first order equations. (Hint: let $y_1 = x$, $y_2 = x'$ and $y_3 = x''$. Try to write an equation for $y_3' = x'''$.) Write an ODE file for this with parameters, $a = 1, b = 2, c = 3.7$ and initial data, $x = 1$,$x' = 0$ and $x'' = 0$. Integrate this for a long time to see the chaotic orbit. Set the `maxstor` option to something like 20000 (by adding the line `@ maxstor=20000` somewhere in your ODE file) and integrate for a total of 400. You may want to look at $x(t)$ versus $x'(t)$ (y_1 versus y_2).

5. Write an ODE file for the Rossler attractor:

$$\begin{aligned} x' &= -y - z \\ y' &= x + ay \\ z' &= bx - cz + xz \end{aligned}$$

with parameters $a = .36$, $b = 0.4$, and $c = 4.5$ and initial data, $x = 0$, $y = -4.3$, and $z = 0$. Run this file for a total of 200 point. Make a three-dimensional plot by clicking on the little boxes next to the three variables in the **Initial Data Window** and then click on the `xvsy` button in the **Initial Data Window**.

6. Write and simulate the following differential equation which has 3 limit cycles:

$$\begin{aligned} x' &= xf(r) - y \\ y' &= yf(r) + x \\ f(r) &= (1 - r)(2 - r)(3 - r) \\ r &= \sqrt{x^2 + y^2} \end{aligned}$$

Window it as $[-4, 4] \times [-4, 4]$. Can you create a differential equation with 5 limit cycles?

7. As a bonus problem, run some of the examples in the text and that you have written yourself to see what they do.

3.5 Discontinuous differential equations.

3.5.1 Integrate-and fire models

Consider the integrate-and-fire model for a neuron

$$\frac{dV}{dt} = I - V$$

with the "reset" condition that each time V hits some value V_T it is reset to 0. This leads to an oscillation when $I > V_T$. Often these models are coupled to each other through "alpha" functions, $\alpha(t) = b^2 t \exp(-bt)$. Thus, the pair satisfies:

$$\frac{dV_1}{dt} = I - V_1 + gs_2(t) \qquad \frac{dV_2}{dt} = I - V_2 + gs_1(t)$$

where each time, t^*, V_j crosses V_T, it is reset to 0 and the quantity $\alpha(t - t^*)$ is added to s_j. *XPPAUT* has a mechanism for incorporating such discontinuities: **global flags**. These are quantities that the integrator checks for zero crossings. If a zero crossing occurs, then the variables can be updated and thus discontinuously changed. The integrators are restarted so that even the adaptive step solvers will work fine. Here is the ODE file for a single integrate and fire oscillator, `iandf1.ode`:

```
# integrate and fire model
v'=-v + I
global 1 v-vt {v=0}
par I=1.2,vt=1
@ dt=.01,ylo=0
done
```

The new line is `global 1 v-vt {v=0}`. This asserts that there is a global flag to check. The general syntax for flags is

```
global sign condition {event1;event2;...}
```

The `condition` is evaluated at each time step. If the `sign` is 1 and the condition changes sign from negative to positive or if the `sign` is -1 and the condition changes sign from positive to negative, then each of the events is done. If the `sign` is 0, then the event occurs only if the condition is identically zero. Events always have the form `x=expr` where `x` is one of the *variables* and `expr` is some expression. Thus, in the above ODE file, if $V - V_T$ changes from negative to positive (ie, V crosses threshold from below) then V is reset to 0. The line `@ dt=.01,ylo=0` sets the timestep to be 0.01 and the minimum value of the y-axis to be zero. Run this file and integrate the equations an you will see a nice oscillation. Move the mouse inside the graphics window and hold the left button down while moving the mouse. At the bottom of the main window, the coordinates of the mouse are shown. You can use this to compute the period of the oscillation; about 1.8.

Let's write an ODE file for the coupled system. The question is how to implement the alpha function. Since $b^2 t \exp(-bt)$ is a solution to

$$s'' + 2bs' + b^2 s = 0 \quad s(0) = 0 \quad s'(0) = b^2$$

this suggests letting s_j evolve according to this equation with the condition that each time the voltage crosses threshold, $s_j'(t)$ is incremented by b^2. Here is the ODE file, `iandf2.ode`:

```
# two integrate and fire models coupled with alpha functions
#  b^2*t exp(-b*t)
#  we solve these by solving a 2d ode
v1' = -v1 + i1 + g*s2
v2' = -v2 + i2  + g*s1
s1'=s1p
s1p'=-2*b*s1p-b*b*s1
s2'=s2p
s2p'=-2*b*s2p-b*b*s2
#
init v1=.1
par i1=1.1,i2=1.1,vt=1,g=.2
par b=5
global 1 v1-vt {v1=0;s1p=s1p+b*b}
global 1 v2-vt {v2=0;s2p=s2p+b*b}
@ dt=.01,total=100,transient=80
@ xlo=80,xhi=100,ylo=0,nplot=2,yp2=v2
done
```

Here are some comments on the ODE file

1. We rewrite the second order equations for s as a pair of first order ODEs.
2. We initialize $V_1 = .1$ so that it is not identical to V_2 to break the symmetry.
3. Each time V_1 crosses V_T from below, V_1 is reset to 0 and s_1' (`s1p`) is incremented by b^2. A similar condition is set for V_2.
4. I have set a whole lot of options so that the user doesn't have to worry about them and can just run the integration. I have set the integration time step to 0.01, the total integration to 100 time units, and I only will start storing data after a transient of 80 time units. I set the low and high limits of the x-axis to 80 and 100 respectively. I set the lower limit of the y-axis to 0. I tell *XPPAUT* to plot 2 curves in the window and tell it the second curve to plot is V_2. (Note, by default, the x component of a plot is time, t and the y component is the first variable defined in the file.)

Run this equation in *XPPAUT* and note that both V_1 and V_2 are synchronized. Change the initial conditions however you want (but make sure that $V_j(0) < 1$) and the solution will always go to synchrony. Change b to 1 and integrate again. The oscillations alternate; synchrony is not stable. Now change b back to 5 and change g to -.2. Integrate again. Notice that synchrony is unstable. Now, change b to 1 again. Set all variables to 0 except V_1. Set $V_1 = .1$ and integrate. You should see synchronous oscillations. Set $V_1 = .4$ and integrate, the oscillations alternate. This result was proven by van vreeswijk et al (1994) (See Figure 3.2.)

3.5.2 Clocks - regular and not

There are many other examples of models that involve discontinuous right-hand sides. The simple clock is an excellent example. We model a pendulum clock as follows. It consists of a decaying sinusoid that receives a kick each time it reaches its maximum on the left-hand side. Let x be the angle of the pendulum and y be the velocity. We model the decaying sinusoid as

$$\frac{dx}{dt} = -ax - by \quad \frac{dy}{dt} = -ay + bx$$

with the kick occurring whenever $y = 0$ and $x < 0$. The kick is toward the left with magnitude k. Thus when $y(t) = 0$ and $x(t) < 0$ then $x(t) = x(t) - k$. The ODE file is

```
# the clock model - a linearly decaying spiral that is kicked out
# by a fixed amount
x'=-a*x-b*y
y'=-a*y+b*x
# heres the kicker
global -1 y {x=x-k}
par a=.1,b=1,k=.5
init x=.5,y=0
# change some XPP parameters to make a nice 2D phase-plane
@ total=150,xlo=-1.5,ylo=-1,xhi=1.5,yhi=1,xp=x,yp=y
done
```

As usual, I have set some internal parameters so that you are looking at a phase-plane projection. Let's take a look at the `global` declaration. The linear equation (without the kick, this is a linear equation, but since the kick is dependent on the values of the variable, the ODE is nonlinear) is a spiral source and since $b > 0$ the flow is counter clockwise. Thus, when $y = 0$ and $x < 0$ the variable y crosses 0 from positive to negative. Thus the choice of `-1` in the `global` statement: this means crossing from above to below. When this happens, x is kicked backwards by the amount k. Run this program. Change the initial conditions - note that no matter what you choose, the solutions always converge to a stable periodic solution. Let's add another window and plot x and y as a function of t. Click on `Makewindow Create` (**M C**). The main window will be repeated. Note the little rectangle in the upper left corner. This indicates that this view is the active view. All new plotting and changes in the graphics will occur in this window. If you want the main window to be active, simply click in it. For now, make the little window active and stretch it out a bit. In the **Initial Data Window** , click on the little boxes next to the `x,y` variables and then click on the button `xvst`. The result of this is that x and y will be plotted with x white and y red. Click on `Window/zoom Zoom in` (**W Z**) to zoom in on this and pick a few cycles so that you can see it better by clicking with the mouse on the little window and releasing the mouse after choosing a suitable area. If you mess up, click on `Window/zoom Fit` (**W F**) which will resize the window to contain all the data. The period of this clock is about 6.3 or so.

A chaotic clock.

We can make this little clock chaotic and in so doing introduce some additional features of *XPPAUT*. In fact, it is easy to prove that the resulting system is chaotic. Here is what to do. Change a from 0.1 to -0.1 and change the kick, k from 0.5 to -0.5. Using the initial conditions $x = 0.5$ and $y = 0$ you should see a chaotic solution as in Fig 3.3. You can prove this is chaotic (this is left as an exercise), the key point is that since $a < 0$ the damping is negative so that nearby trajectories will exponentially diverge. The kick to the right keeps the solutions bounded. However, if you start with x large enough, then the solutions will grow without bound. You can also prove that there is an unstable periodic solution with initial conditions at about $(x, y) = (0.7831, 0)$. Any initial data inside this unstable periodic will converge to the chaotic attractor. Integrate using as initial conditions the values at the end of the last integration (click on `Initialconds Last` , or **I L** .) Now let's look at the maximal Liapunov exponent, a measure of chaos. We haven't gotten much data, but can try it anyway. Click on `nUmerics stocHastic Liapunov` (**U H L**) and after a brief moment, a window will come up telling you an approximation of the maximal exponent. The approximation is not very good (the actual value is 0.1) but if you integrate longer, you will get a better value. (Try this: in the numerics menu, change `Total` to 1000. Change `nout` to 5. Integrate. Recompute the Liapunov exponent and it will be much closer to 0.1.) Another quantity that is indicative of chaos is the power spectrum. To get this, click on `nUmerics stocHastics Power` (**U H P**) and choose y as the variable to transform. In the **Data Viewer** the two columns formerly occupied by x and y are now filled with the power and the phase. (That is, the FFT returns the cosine and sine coefficients of the spectrum, (c, s); the power, $p = \sqrt{c^2 + s^2}$ is the magnitude and the phase, $\phi = \arctan(s/c)$ is the angle.) The column occupied by t is replaced by the frequency. You can plot the spectrum by graphing `x` against `t`. Note the broadband frequency. There are solitary peaks at about 318, 636, and so on corresponding to the unstable periodic orbit. You can get back the original orbit by clicking on the `nUmerics Stochastic Data` entry and then tapping **Esc** to get back to the main menu.

There are a number of other interesting ways to analyze chaotic data. We will return to these at a later time.

3.5.3 The dripping faucet

One of the classic examples of chaos in a physical system is the dripping faucet. Bob Shaw (1985) collected data on the interval between drips of the faucet. He found through a variety of analyses that the process was chaotic and simulated it on an analog computer. The model consists of a mass on a spring. The mass grows linearly and when the spring extends beyond a certain limit, a fraction of the mass drops off. The fraction is proportional to the velocity. The equations are:

$$\frac{d}{dt}(m\frac{dx}{dt}) = mg - (\mu\frac{dx}{dt} + kx) \quad \frac{dm}{dt} = f,$$

with the condition that if $x(t)$ crosses 1 then the mass is decreased by an amount that depends on the velocity, $v = dx/dt$. Thus, $m(t) \to (1 - hv(t))m(t)$. The ODE file for this model is

```
# faucet.ode
x'=v
v'=g-(k*x+(f+mu)*v)/m
m'=f
global 1 x-1 {m=max(m-h*m*v,m0)}
par f=.4,g=.32,h=4,m0=.01,mu=.6,k=1
init x=0,v=0,m=1
@ total=200
done
```

The first three equations are straightforward enough; we have just rewritten the second order system into a pair of first order systems and used the fact that $(mv)' = m'v + mv'$. The `global` declaration states that when x crosses 1 from below, we decrement m by hmv. The additional function `max(x,m0)` makes sure that the mass never falls below some minimum value or becomes negative. Run this and look in the `(x,v)` phaseplane to see something very similar to the Rossler attractor (see the exercises in the previous section). Play with the parameter, f and see a variety of different solutions and bifurcations.

3.5.4 Exercises

1. Write a differential equation for John Tyson's cell growth model:

$$\begin{aligned} u' &= k_4(v-u)(a+u^2) - k_6 u \\ v' &= k_1 m - k_6 u \\ m' &= bm \end{aligned}$$

   where $k_4 = 200, k_1 = 0.015, k_6 = 2, a = 0.0001, b = 0.005$. m is the mass and when the variable u drops **below** 0.2 the cell divides and $m = m/2$. Start with $u(0) = .0075, v = 0.48, m = 1$. Integrate this for a total of 1000 with a timestep of 0.25 and using the `DoPri(8)` integrator. Plot the mass as a function of time. Notice that after a few transients, it goes to a periodic solution.

2. Consider the constant planar vector field:

$$x' = 1, \qquad y' = a.$$

   If this is integrated on a torus; that is, each time $x = 1$ (respectively, $y = 1$), x (respectively, y) is reset to 0, then if a is irrational, the trajectories densely fill the unit square representing the unfolded torus. For the first part of this exercise, verify this with a numerical simulation. Plot the phase-plane for the unit square and use the `Graphics Edit curve 0` to change the `Linetype` to zero so that just dots are drawn. Use the `global` declaration to reset x and y.

   Curiously enough, if you solve this equation on a Klein bottle instead of a torus, all solutions are periodic! How do you make a Klein bottle phase space? Recall that to create a Klein bottle, you must reverse the orientation along one of the edges of the square. Thus, when $y = 1$, reset y to zero *and* set $x = 1 - x$ which flips its orientation. Make an *XPPAUT* file that does this and verify that all solutions are periodic. Prove your answer.

3. Put a lemon seed in a glass of soda. It will begin to sink, but as it sinks, bubbles accumulate on it making it more buoyant. It will slow down and float toward the surface. When it hits the surface, the bubbles pop and it begins to sink again. We can model this in a manner similar to the dripping faucet. As the seed sinks, it accumulates bubbles which have considerably less density than the soda. Eventually the volume fraction of the bubbles overcomes the seed density and the seed floats up. The difference between the seed density and the density of the soda water provides the driving force. We assume that the soda is sufficiently viscous so that we can ignore inertial effects. Once the seed hits surface, the bubbles pop; more so if the velocity high. Let V_0 denote the volume of the seed with no bubbles and let $d > 1$ be the density with the density of the soda water equal to 1. Let $V(t)$ be the volume of bubbles accumulated and we will let their density be 0. The density of the seed is

$$d_s = d\frac{V_0}{V_0 + V(t)}$$

The velocity of the seed is

$$x' = k(d_s - 1)Q(\epsilon - x).$$

The function Q prevents the seed from rising above the soda water and thus it is kept below ϵ. We will use the step function for Q. The accumulation of bubbles is

$$V' = c$$

where c is just a constant. Let $f(\dot{x})$ denote the fraction of bubbles that remains once the seed hits the surface. Then when $x = 0$, we replace $V(t)$ by $f(\dot{x})V(t)$. A linear f seems to be a good approximation:

$$f(\dot{x}) = \max(f_0 - f_1\dot{x}, 0)$$

Try writing an ODE file and simulating it. I made a nice seed oscillator with $\epsilon = 0.1$, $f_0 = 0.05$, $k = V_0 = 1$, $d = 2$ and $c = 0.1$..

4. **Project idea.** The drinking duck is a famous toy which consists of a glass tube filled with freon and decorated to look like a duck. When the duck's head is moistened, evaporative cooling causes the freon to rise up to the duck's head and this makes the duck swing down into a waiting glass of water. This action moistens the duck's head and also causes the freon to drop back to the duck's belly. He swings back and the process repeats. Model this as a pendulum with two weights. The head weight grows slowly proportional to the angular velocity. When the head drops, the weight is reset to the bottom and the process begins anew.

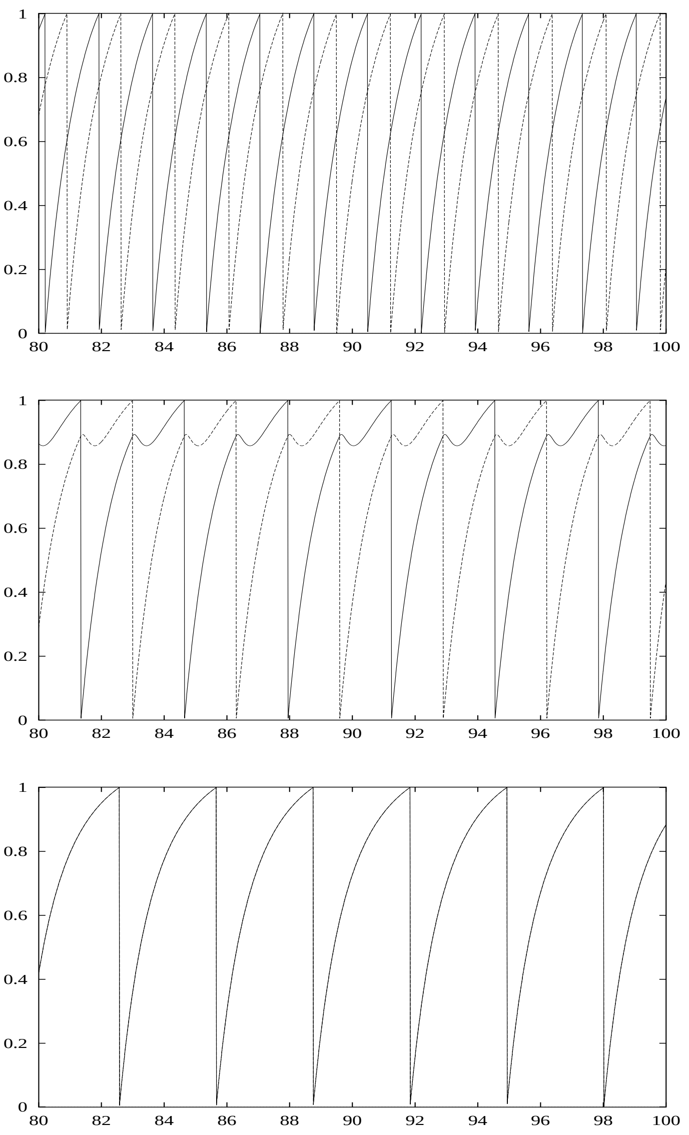


Figure 3.2. *A pair of integrate-and-fire neurons coupled with alpha-function synapses. With slow excitatory coupling synchrony is unstable; with fast inhibition, synchrony is also unstable; with slow inhibition, synchrony is stable.*

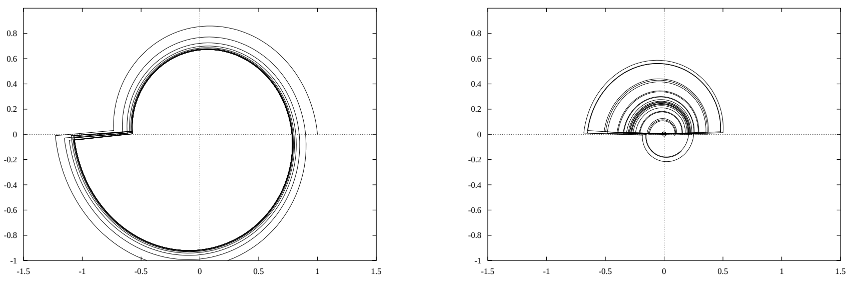
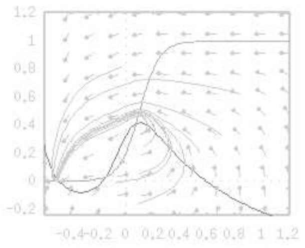


Figure 3.3. *A kicked clock and its chaotic brother.*



Chapter 4

XPPAUT in the classroom

XPPAUT , although developed as a research tool has been used by many people in a variety of courses in order to allow students to look at models and differential equations on the computer. Since the program is free and runs on a variety of platforms, it is suitable for classroom use. While the interface is not as simple as some dedicated systems such as Java applets or the Macintosh program, MACPHASE, it is flexible and with a little thought by the instructor, it becomes relatively easy to write ODE files illustrating all the standard figures in most textbooks. I have used *XPPAUT* with many differential equations and modeling texts at the undergraduate and graduate levels. Some of these are specialty courses like Computational Neuroscience or Quantitative Cardiac Physiology. However, I will not delve into such specialized applications in this chapter. Rather, you can go to the WWW and see some examples from various advanced courses. I will start with some simple examples of using *XPPAUT* to create plots in two and three dimensions without reference to differential equations at all. I will then consider a variety of topics in first order discrete dynamical systems. I will illustrate how to make cobweb diagrams, draw bifurcation diagrams, compute rotation numbers, and diagrams of the maximal Liapunov exponent. I will then demonstrate how to use *XPPAUT* to make Julia sets and the famous Mandelbrot set. In the next subsection, I will discuss one-dimensional non-autonomous differential equations and finally planar dynamical systems.

4.1 Plotting functions.

Suppose that you want to plot the function of one variable $f(x)$ from x_0 to x_1. Then the following ODE file will do the trick:

```
# plot1.ode
# plot f(x)
par xlo=-2,xhi=2
f(x)=x*(1-x^2)
s'=1
```

```
x_=xlo+s*(xhi-xlo)
aux y=f(x_)
aux x=x_
@ xp=x,yp=y
@ xlo=-2,xhi=2,ylo=-4,yhi=4
@ total=1.001,dt=.01
done
```

Most of this file is to define the plotting area and numerics. To change the ranges of plotting, just vary the parameters `xlo, xhi` and to change the function, just click on `File Edit Functions` and change the function. To keep the original graph and subsequent graphs, click on `Graphics Freeze On freeze` and up to 26 curves can be kept.

Here is an example in which the first 5 terms of the Taylor series for the function $\sin(x)$ are plotted on the interval $[-\pi, \pi]$:

```
# sintayl.ode
# first 5 terms of the Taylor series for sin
z1=x
z2=z1-x^3/6
z3=z2+x^5/120
z4=z3-x^7/5040
z5=z4+x^9/362880
x'=2*pi
init x=-3.1415926
aux y=sin(x)
aux y[1..5]=z[j]
@ total=1,dt=.005
@ xlo=-3.15,xhi=3.15,ylo=-1,yhi=1
@ nplot=6,xp=x,yp=y
@ yp[2..6]=y[j-1],xp[j]=x
done
```

NOTE. By defining a series of internal variables, `z1, ...` I can just add the required new terms sequentially. The last line tells *XPPAUT* that all these quantities should be plotted. They will appear as different colored curves.

You can easily use *XPPAUT* to make polar coordinate plots of the form $r = f(\theta)$ (or $\theta = f(r)$):

```
# polarpl.ode
# polar plots
f(z)=1+cos(z)
theta'=1
init theta=-12
r=f(theta)
aux x=r*cos(theta)
aux y=r*sin(theta)
@ total=25
@ xp=x,yp=y,xlo=-2,xhi=2,ylo=-2,yhi=2
done
```

There is a parameter a so that one could plot this over a range of values of a. Edit the function to get some other polar curves.

Other parametric plots are just as easy to plot. For example, this plots $(x(t), y(t))$ in the plane:

```
# parametric.ode
f(t)=cos(a*t)
g(t)=sin(b*t)
par a=4,b=1
s'=1
aux x=f(s)
aux y=g(s)
@ xp=x,yp=y,xlo=-2,ylo=-2,xhi=2,yhi=2
@ total=50
done
```

Similarly, three-dimensional parametric plots are also possible. Here is one where I have set up all the axes. Change parameters and plot away.

```
# param3d.ode
# 3d parametric plots
f(t)=cos(a*t)
g(t)=sin(b*t)
h(t)=sin(c*t+d)
par a=4,b=1,c=2,d=.75
s'=1
aux x=f(s)
aux y=g(s)
aux z=h(s)
@ xp=x,yp=y,zp=z,xlo=-2,ylo=-2,xhi=2,yhi=2
@ axes=3d
@ xmax=1.25,ymax=1.25,zmax=1.25,xmin=-1.25,ymin=-1.25,zmin=-1.25
@ total=50
done
```

As a last example of plotting in three-dimensions, here is a nice trick in which I plot the three-dimensional parametric plot along with a two-dimensional projection below it:

```
# param3d2.ode
# 3d parametric plots with 2d projection
f(t)=cos(a*t)
g(t)=sin(b*t)
h(t)=sin(c*t+d)
par a=1.34,b=1.12,c=2.28,d=.75
```

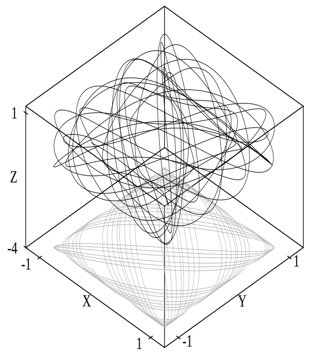


Figure 4.1. *Three-dimensional plot illustrating a projection on the coordinate plane*

```
par zlo=-4
s'=1
aux x=f(s)
aux y=g(s)
aux z=h(s)
aux zplane=zlo
@ xp=x,yp=y,zp=z,xlo=-2,ylo=-2,xhi=2,yhi=2
@ axes=3d
@ xmax=1.25,ymax=1.25,zmax=1.25,xmin=-1.25,ymin=-1.25,zmin=-4
@ nplot=2,xp2=x,yp2=y,zp2=zplane
@ total=100
done
```

I have added another plot in which I keep the z-value constant. This is a useful trick and lets one plot a variety of interesting projections against the coordinate planes. Only `zmin` and the parameter `zlo` need to be changed in order to change the position of the planar projection. By plotting z, y and holding x at some fixed value, you can similarly plot the (y, z) projection. See figure 4.1 for the result.

4.2 Discrete dynamics in one-dimension.

Many discussions of dynamical systems begin with the study of one-dimensional maps. These have the form:

$$x_{n+1} = f(x_n).$$

One of the standard ways to illustrate the dynamics of these maps is to draw a cobwebbing diagram. This is a plot of x_{n+1} versus x_n along with the line $y = x$ and the function $y = f(x)$. *XPPAUT* does not have a specific cobwebbing function built into it, but it is very easy to fool the program into making a cobweb plot. Every other point shares either the same x value or the same y value, thus the cobweb plot consists of a series of horizontal and vertical lines. Once this is done, you also want to plot the function and the line $y = x$. Here is an example cobweb map for the logistic function:

```
# cobweb.ode
# way to fool XPP into cobwebbing
# first I define a function that every other step evaluates
# the map -- in the alternate steps, it just keeps the same
# value so that it alternates between horizontal and vertical
# jumps
g(x,y)=if(mod(t,2)<.5)then(f(x))else(y)
# note that 't' is the iteration number 0,1,2,...
# if t is even evaluate f otherwise keep the old y
y(t+1)=g(x,y)
x(t+1)=if(t==0)then(x)else(y)
# note that x(t+2)=f(x(t)) so every other point is the map!
f(x)=a*x*(1-x)
par a=3.95
# these are just useful for plotting f(x),x
par xlo=0,xhi=1,nit=25
#
init y=0,x=.25
# always start y=0
#
# here I create scaled x-values to plot f(x)
xx=xlo+(xhi-xlo)*t/nit
aux map=f(xx)
aux st=xx
# some convenient settings for the graphics
@ xlo=0,ylo=0,xhi=1.001,yhi=1.001
@ xp=x,yp=y
@ nplot=3
# add the plots y=x and y=f(x)
@ xp2=st,yp2=st
@ xp3=st,yp3=map
# tell xpp that it is discrete and iterate 25 times
@ meth=discrete,total=25
done
```

This may seem a bit involved but it works and it is very easy to adapt it to other functions. Simply edit the function $f(x)$ and perhaps the ranges of the map `xlo,xhi`. Run this for a variety of values of $a < 4$. Try the map $f(x) = \mathrm{mod}(x + 3/4 + a\sin 2\pi x, 1)$ for $a < 1/2$.

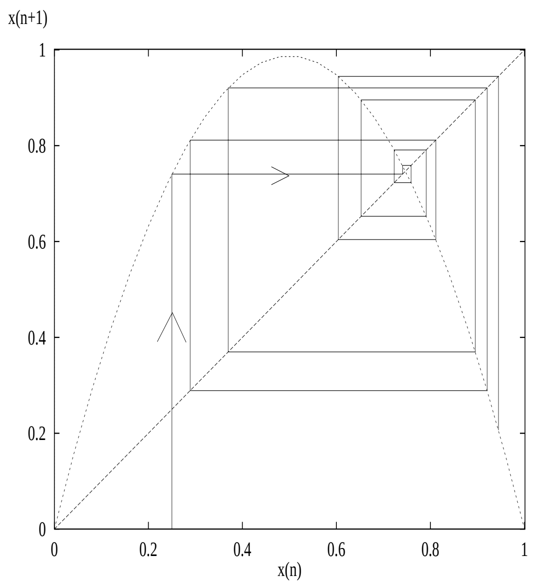


Figure 4.2. *A cobweb plot of the logistic map*

4.2.1 Bifurcation diagrams

Another classic picture that is shown in many books is the bifurcation diagram for the logistic map. This is done by plotting the parameter a along the $x-$axis and iterates (after transients) of the map along the $y-$axis. To do this in *XPPAUT* , you want to make the parameter a variable and then use the Range Integration option to draw the diagram. Here is the *XPPAUT* file

```
# logbif.ode
# the logistic map bifurcation diagram
f(x)=a*x*(1-x)
x'=f(x)
a'=a
init x=.1
init a=2
@ maxstor=100000,total=500,trans=350,meth=discrete
@ xlo=2,xhi=4.001,ylo=0,yhi=1.001,xp=a,yp=x
done
```

NOTES: I have set `maxstor=100000` so that *XPPAUT* will store many points. I throw away the first 350 iterates to avoid transients. I treat the parameter as a variable so it can be plotted; parameters are not generally allowed to lie on axes.

Run this file and then do the following. First click on `Graphics Edit` and select `0`. Then in the resulting dialog box, change the line type from 1 to 0. This makes *XPPAUT* just plot points rather than lines. Now we can draw the diagram. Click on `Initialconds Range` and fill in the dialog box as follows:

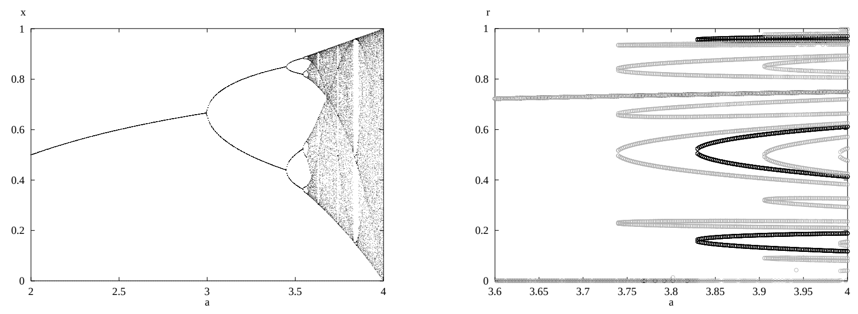


Figure 4.3. *The logistic map showing an orbit diagram on the left and a plot of the periodic orbits of period 3 and period 5 on the right.*

`Range over:` a
`Steps:` 200
`Start:` 2
`End:` 4
`Reset storage:` N

and click on **Ok** . You should see the bifurcation diagram appear. You can put more points in it by changing the number of steps from 200 to 500. The `Range` command allows you to range over either parameters or initial data and save the results. Figure 4.3 shows the orbit diagram we just computed.

4.2.2 Periodic points

The "bifurcation" diagram constructed above is actually more correctly an orbit map. That is, it simply is a record of the *stable* solutions for any particular choice of parameters. A true bifurcation diagram should show solutions which are both stable and unstable. For example, one can ask where all the period three points are for the map. Period three is notable since it implies that there are periodic points for every possible period. This is one of the standard measures of chaotic behavior. How can one find the periodic points of a map as a function of a parameter? Consider a fixed value of the parameter and the third iterate of the map. Since the periodic points are not necessarily stable (and in fact are unstable), we need a method of finding them that is independent of stability. Newton's method provides a good way to search for them as the convergence of the iterates is independent of the stability. Thus, for a fixed parameter value, we could sweep through a range of starting values and plot those for which the Newton's iterates converge. This shotgun approach has the advantage that we can pick up new branches of orbits as the parameter changes. In *XPPAUT* , there is a way to run a two-parameter range of simulations. This allows us to sweep through both the parameter and the state space. Given the map $f(x)$, a point of period p satisfies $f^p(x) = x$. (Here, $f^2(x)$ means, $f(f(x))$ and so on.) Let $g(x) = x - f^p(x)$. We need to find roots of $g(x)$. Newton's method provides an iterative scheme:

$$r_{n+1} = r_n - g(r_n)/g'(r_n).$$

This rapidly converges if we have a close-by starting guess. But by sweeping over many values of r we are almost assured of catching all of them. Since we are solving for the roots numerically, we want to make sure that convergence to a true zero has occurred. Thus, if $|g(r)|$ is less than a specified tolerance, we will assume that we have found a root. Finally, we compute $g'(r)$ numerically to simplify the calculation. With these preliminaries, here is an *XPPAUT* file that will let you compute periodic points of any specified period:

```
# logper.ode
# finds the periodic points of the logistic map
#
f(x)=a*x*(1-x)
# recursively define pth iterate
ff(x,p)=if(p<=0)then(x)else(ff(f(x),p-1))
# find zeros of g
g(x)=x-ff(x,p)
#
q=g(r)
# Newtons method with numerical derivative
r'=r-eps*q/(g(r+eps)-q)
a'=a
# if within tol of root, then OK
aux err=tol-abs(q)
par p=3,eps=.000001,tol=1e-7
@ meth=discrete,total=20,maxstor=50000
@ xp=a,yp=r,xlo=3,xhi=4.0001,ylo=0,yhi=1.00001
done
```

NOTES: I have defined the p^{th} iterate of f recursively. The main iteration is the implementation of Newton's method. I define a fixed variable `q` since I need to refer to it three times and thus the computation is sped up. The parameter `tol` gives the tolerance that I require for a root. The parameter `eps` is for computing the numerical derivative

$$g'(x) \approx \frac{g(x+\epsilon) - g(x)}{\epsilon}.$$

Let's use this to compute the period-three bifurcation curves. Run *XPPAUT*. Click on `Graphics Edit` (**G E**) and choose curve `0`. Change the linetype to `0` so that dots will be drawn and click **Ok** . Next, click on `Numerics Poincare map Section` . Fill in the dialog box as

`Variable:` err
`Section:` 0
`Direction:` 1
`Stop on section:` y

This tells *XPPAUT* to only plot the points once the quantity `err` crosses zero. This means that the value of the root is within the specified tolerance. It also guarantees that Newton iterates which do *not* converge to a root will not be plotted. Now click on `Esc` to exit the numerics menu. We now will set up the two-parameter range integration. Click on `Initialconds 2 Par range` (**I 2**) to get a rather unwieldy dialog box. Fill this in as

`Vary1:` R	`Reset storage:` N
`Start1:` 0	`Use old ic's:` Y
`End1:` 1	`Cycle color:` N
`Vary2:` a	`Movie:` N
`Start2:` 3	`Crv(1) Array(2):` 2
`End2:` 4	`Steps2:` 200
`Steps:` 50	`:`

This dialog tells *XPPAUT* which pair of parameters you want to vary. The entry `Crv(1) Array (2)` allows you to vary the two parameters together or independently. Click on `Ok` and the period 3 points will be plotted. If you want to compute at a finer resolution, increase the number of steps for the bifurcation parameter (`Steps2`). If you compute for period 2 points, then you should also increase the number of steps for the initial guesses (`Steps2` as there are many such points and you want to get them all. To save these points and superimpose a diagram with other periodic points, click on `Graphics Freeze Freeze` (**G F F**) and choose `-1, -2, ..., -9` for the `color` entry to make *XPPAUT* use points instead of lines. Figure 4.3 shows the period three and five points.

4.2.3 Liapunov exponents for 1-d maps

One of the hallmarks of chaos is the local exponential divergence of nearby points under the dynamics. For one-dimensional maps, this is easy to measure. One needs to look at the average of the logarithm of the absolute value of the derivative of f evaluated at a point on the attractor. Thus, for a one-dimensional map, the maximal Liapunov exponent is

$$\lambda = \lim_{N\to\infty} \frac{1}{N} \sum_{j=1}^{N} \ln|f'(x_j)|.$$

If $\lambda < 0$ then the attractor is asymptotically stable. If $\lambda = 0$ the attractor is a periodic orbit. Finally if the attractor is chaotic, then $\lambda > 0$. For the logistic map with $a = 4$ it is possible to show that $\lambda = \ln 2 \approx 0.693147$. The following *XPPAUT* file illustrates how to compute this exponent as a function of the parameter. I keep a running sum of the log of the derivative after throwing out the first 100. I compute the running average of this sum and keep only the final value computed after 2000 iterates. Here is the file

```
# logliap.ode
# the liapunov exponent of the logistic map
f(x)=a*x*(1-x)
fp(x)=a*(1-2*x)
init x=.1,a=2,z=0
x'=f(x)
a'=a
z'=z+heav(t-100)*ln(abs(fp(x))+1e-8)
aux liap=z/max(t-100,1)
@ xlo=2,xhi=4,ylo=-2,yhi=1
@ total=2000,trans=2000
@ meth=disc,bound=1000000
@ xp=a,yp=liap
done
```

NOTES: The statement `z'=z+heav(t-100)*ln(abs(fp(x))+1e-8)` serves two purposes. First, if $f' = 0$, the addition of the small amount 1e-8 prevents a singularity in the log. Secondly, premultiplying by `heav(t-100)` only adds the numbers after 100 iterations. The quantity `liap` is just the running average of `z` and it the quantity of interest. `@ total=2000,trans=2000` tells *XPPAUT* to iterate for 2000 and only keep the last point. Run this and integrate using the range option. Use the same parameters in this as you did in computing the bifurcation diagram.

There is an easier way to compute Liapunov exponents over a range of parameters. *XPPAUT* has a built-in algorithm for the computation of Liapunov exponents for differential equations and maps. It works similarly to the idea described above. Load up the logistic equation:

```
# logistmap2.ode
x(t+1)=a*x*(1-x)
par a=3
init x=.54321
@ total=1000,trans=500,meth=disc
done
```

I have set the total iterates to 1000 and thrown away the first 500. Once you have *XPPAUT* up and running click on `nUmerics stocHastic Liapunov` and choose `1` to look at a range. Choose `a` as the parameter to range over, choose `500` as the number of steps, and choose `3` and `4` as the start and end. Click on `Ok` and after a short wait, the calculation will be done. Click on `Escape` to go to the main menu and then click on `Xi vs t` and hit `Enter`. You will get a nice plot of the maximal exponent as a function of the parameter a. Note that when $a = 4$, $\lambda_{Max} = 0.6911265$ which is close to the actual value of $\ln(2) = 0.69314$. The difference is due to the finite number of points computed. See figure 4.4.

4.2.4 The Devil's staircase

If you periodically force an oscillator, then a number of interesting phenomena can occur. One of these is phase-locking in which the resulting behavior is periodic. If

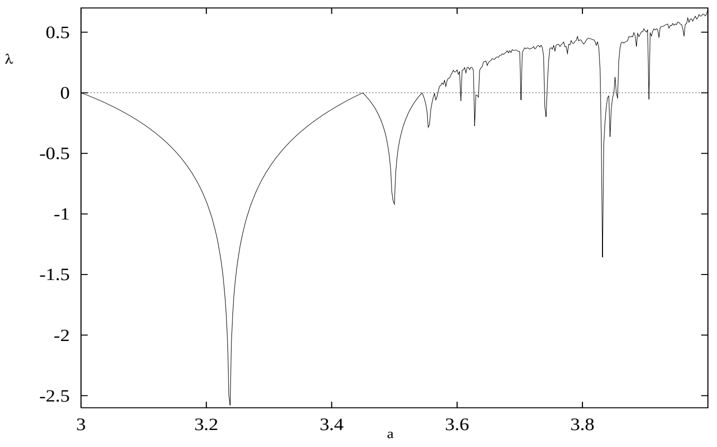


Figure 4.4. *Maximal Liapunov exponent as a function of the parameter a for the logistic map.*

the forced oscillator cycles N times while the forcing function cycles M times, this is called $N : M$ locking. One way to model a periodically forced oscillator is to look at a one-dimensional map

$$x_{n+1} = f(x_n)$$

where the function, f takes the unit circle back to itself. x_n is the phase of the oscillator after the n^{th} stimulus. The standard map has the form:

$$f(x) = x + b + a\sin(2\pi x) \bmod 1.$$

Consider $a = 0$ for a moment. Suppose that $b = N/M$. Then after M stimuli, $x_n = N + x_{n-M}$, that is x has traversed N cycles while the stimulus has traversed M. This is $N : M$ locking. However, it is not very robust. Consider the ratio,

$$R_n = \frac{x_n}{n}.$$

As $n \to \infty$, clearly, $R_n \to N/M \equiv \rho$. This limiting value, ρ is called the rotation number. Now, suppose that $a \neq 0$. Then we can no longer explicitly write down the solution x_n. However, the rotation number still exists (if $f'(x) > 0$, e.g. if $|a| < 1/(2\pi)$) and in fact depends continuously on the parameter b. This is a consequence of Denjoy's Theorem (p. 301, Guckenheimer & Holmes). Furthermore, it is constant almost everywhere. Thus, if we plot ρ as a function of b, we will get what is known as the *Devil's Staircase.* We can use *XPPAUT* to plot the rotation number as a function of the parameter b to see what this looks like. Here is the *XPPAUT* file:

```
# rotnum.ode
# the rotation number for the std map
```

```
f(x)=x+b+a*sin(2*pi*x)
x'=f(x)
b'=b
par a=.15
init b=0
init x=0
aux rho=x/max(t,1)
@ bound=100000
@ total=1000,trans=1000,meth=disc
@ xp=b,yp=rho,xlo=0,ylo=0,xhi=1.001,yhi=1.001
@ rangeover=b,rangestep=200,rangelow=0,rangehigh=1,rangereset=no
done
```

NOTES: As with the other bifurcation problems, we have treated the parameter as a variable so that it is available for plotting. We do not mod x by 1 here so that we can compute the total accumulation of phase. The rotation number is approximated by the finite ratio, x/t where t is the iteration number. As only the last value is of interest, the `trans=1000` tells *XPPAUT* to only look at the final point. The last line sets up the range integration so that you won't even have to fill in a dialog box.

Run this with *XPPAUT*. Click on `Initialconds Range` (**I R**). The dialog box should look like this:

`Range over:` b
`Steps:` 200
`Start:` 0
`End:` 1
`Reset storage:` N

Click on **Ok** . The Devils staircase will be drawn for you. (If you want a really detailed picture, make the number of steps 2000. Then zoom into different regions and see that there are many flat regions corresponding to constant rotation numbers. If the amplitude parameter is larger than $1/(2\pi) \approx 0.16$, the map is no longer invertible and the assumptions for Denjoy's theorem no longer hold. Change a to 0.4 and redraw the staircase.

4.2.5 Complex maps in one-dimension

The Mandelbrot set has often been called the most complex object in mathematics. While this is likely to be false, the computation of this set has led to a huge number of little applications which will draw it for you to any desired resolution. Many of these applications offer little explanation of the set nor of its mathematical significance. Here, I will offer a brief description of what the set means. First, let's consider the logistic equation that we looked at above:

$$x_{n+1} = ax_n(1 - x_n).$$

Set $a = 4$ and make the following transformation: $u = 2x - 1$. Then

$$u_{n+1} = 1 - 2u_n^2.$$

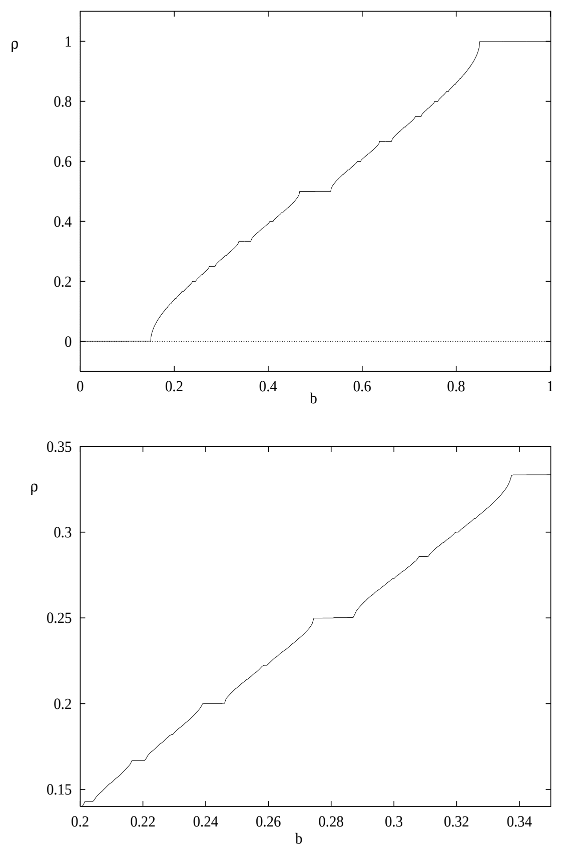


Figure 4.5. *Rotation number for the standard map and a expanded view over a small range of the parameter b.*

Let $z = u + iv$ and consider the complex map:

$$z_{n+1} = z_n^2$$

or in real coordinates,

$$(u_{n+1}, v_{n+1}) = (u_n^2 - v_n^2, 2u_n v_n).$$

Suppose that we restrict z to have a magnitude of 1, so that $u^2 + v^2 = 1$. Then $u^2 - v^2 = 1 - 2u^2$ and we see that the logistic map with $a = 4$ is equivalent to the complex map $z \to z^2$ restricted to the unit circle. The dynamics on the unit circle can best be studied by writing $z = \exp(2\pi i\theta)$ so that the map is now

$$\theta_{n+1} = 2\theta_n \bmod 1.$$

This means that the dynamics is chaotic in almost every sense of the word. If we express a number between 0 and 1 in base 2, then the map just shifts to the left and drops the lowest bit. Every possible periodic solution is found, e.g., 0.100100100... is a period three point. Furthermore, the Liapunov exponent of this is $\ln 2$ since the map expands all points by a factor of 2.

Looking again at the complex map, $z \to z^2$ it is clear that any initial condition that starts inside the unit circle will tend to the fixed point 0 while any initial condition which starts outside of the unit circle will tend to infinity. Thus, the unit circle serves to separate points in the complex plane which are attracted to the fixed point 0 from those that are attracted to the fixed point at infinity. The set of points which are not attracted to these fixed points is called the Julia set. Thus, the Julia set of the map $z \to z^2$ is the unit circle. Now consider the general quadratic map with a complex parameter, c:

$$z_{n+1} = z_n^2 + c.$$

If the parameter c is small then the Julia set of this map should be close to the unit circle. It is a connected set and there is an inside and an outside. However, if c is large, then the Julia set will not look at all like the circle. In particular, there is no separation between initial conditions that stay bounded and those which go off to infinity. The famous Mandelbrot set is the set of all points c for which the Julia set is connected.

Now that we have connected the logistic map to the Mandelbrot set through the Julia set, how can we compute the Julia sets? Since the Julia set is a separatrix between fixed points, we can compute it by starting inside the unit circle and iterating backwards. The inverse of the map $z^2 + c$ is $\pm\sqrt{(}z - c)$. The way we will compute it is to *randomly* pick between the two square roots at each iterate. (Note the idea of randomly choosing one of the square roots makes this computation related to so-called iterated function systems which we visit below.) To compute the square roots, we first write the number $z - c$ in polar form $r\exp(i\theta)$ and then take the square root of r and divide θ by two. Recall that if $u + iv$ is a complex number, then $r = \sqrt{(}u^2 + v^2)$ is its magnitude and $\theta = \tan^{-1}(v/u)$ is its argument. We then write the result of this as a two-dimensional real map. Here is the *XPPAUT* file `julia.ode`

```
# julia.ode
# julia set for z -> z^2+c
par cx=0,cy=0
r=sqrt(sqrt((x-cx)^2+(y-cy)^2))
th=atan2(y-cy,x-cx)/2
s=sign(ran(1)-.5)
init x=.1
x'=s*r*cos(th)
y'=s*r*sin(th)
@ total=2000, meth=disc,trans=100
@ xp=x,yp=y,xlo=-2,xhi=2,ylo=-2,yhi=2
@ lt=0
done
```

NOTES: The function `atan2(v,u)` takes into account the signs of u, v to return the correct argument θ for the number $u + iv$. I have thrown away the first 100 points and iterated it for 2000 times. The last line `lt=0` tells *XPPAUT* that it should not connect the points with lines.

Start up *XPPAUT* with this file. Before iterating, turn off the axes so you don't confuse them with the Julia set: click `Graphics aXes opts` (**G X**) and change the three entries `X-org(1=on)` etc from `1` to `0` thus turning off the axes. Now integrate the equation and you will see an circular Julia set. (It looks elliptical since the graphics view is rectangular.) Change `cy` to -0.5 and integrate the equations again. Try `cy=0.5`. Can you say something about the symmetry of the Julia set? Try `cy=0,cx=-.5`. Try `cy=1.5,cx=0`. Do these look connected? Try `cy=1,cx=0` and iterate for 4000 times. This may or may not be connected! It is hard to tell.

Animated Julia sets. Let's make a whole bunch of Julia sets and animate the result as the parameter varies. We set `cx=0` and vary `cy` from `-1` to `1`. First, change the total number of iterates back to 2000. Click on `Initialconds Range` (**I R**) and fill in the dialog as follows:

`Range over:` cy
`Steps:` 20
`Start:` -1
`End:` 1
`Reset storage:` Y
`Use old ic's:` Y
`Cycle color:` N
`Movie:` Y

Notice that you should type in `Y` in the `Movie` entry since this tells *XPPAUT* you want to save the contents of the window. Also, make sure that no windows other than perhaps the dialog window obscure the main graphics window. Click on **Ok** and 20 Julia sets will be computed. Each screen image is saved in memory so that you can play them back. Now click on `Kinescope Playback` (**K P**) and you can step through each of the plots one at a time by clicking on a mouse button. Click **Esc** to exit. To get a smooth animation of the Julia sets, click on `Kinescope Autoplay` (**K A**). The tell *XPPAUT* how many times you want to repeat the animation (e.g 2) and how many milliseconds between frames (e.g. 100). After this, the animation will be repeated as a smooth movie.

You can save the animations in several ways. One of these requires that you have a separate software program that can encode a series of still images into a movie. An easier way but one which is not quite as efficient is to create an animated GIF. Animated GIF's can be played on most web browsers such as Netscape or Explorer. Click on `Kinescope Make Anigif` and your movie will be replayed. At the same time, a file called `anim.gif` will be produced on your disk. You can view this with `xanim` or other viewing software or just open it with your web browser.

The Mandelbrot set. We have used *XPPAUT* to animate Julia sets. For open sets of parameters the Julia set is connected. The set of parameters, c for which the Julia set is connected is the Mandelbrot set. Another way to look at it is to define it as the set of parameters c such that iterates of $f(z) = z^2 + c$ starting at the origin remain bounded. (Since the Julia set is connected, there is no "escape" to infinity.) This set is typically computed by choosing a parameter, c and iterating a number of times until z_n exceeds some bound. If after this fixed number of iterates, z_n is still bounded, then c is considered to be in the Mandelbrot set. The number of iterates needed to determine whether a point is in the Mandelbrot set is becomes large near the boundary of the set. Often points that are not in the set are color coded according to the number of iterates it takes to become unbounded.

To compute this set in *XPPAUT* , we will use a number of features which control the output and color. Here is the ODE file for computing the Mandelbrot set, `mandel.ode`

```
# mandel.ode
#  drawing the mandelbrot set
#
x'=x^2-y^2+cx
y'=2*x*y+cy
# so that the parameter axes are plottable
cx'=cx
cy'=cy
init x=0,y=0
#
# if this crosses 0 then we are out of the set
aux amp=x^2+y^2-4
#
#
@ xp=cx,yp=cy,xlo=-1.5,xhi=.5,ylo=-1,yhi=1,lt=-1
@ maxstor=40000,meth=disc,total=50
@ poimap=section,poivar=amp,poipln=0,poisgn=1,poistop=1
done
```

Notes. This is a rather simple ODE file with lots of controls added. We could easily set these controls in *XPPAUT* , but for ease of use, set them here. The first two lines are the map to iterate written in the real variables, $z = x + iy$. The next lines are so that *XPPAUT* can use the parameters as coordinate axes. *XPPAUT* only allows auxiliary variables and dynamic variables as axes. The auxiliary variable `amp` changes from negative to positive when the magnitude of z exceeds 2. It can be shown that once this happens, z_n diverges to infinity. The controls in the first line set up the plot axes and bounds. The next line tells *XPPAUT* to allocate room for 40000 points– the default is 4000; the method of integration; and the total iterations per run. We use only 50 since we are only crudely estimating the Mandelbrot set. The last option line starting with `poimap=section` tells *XPPAUT* that we are going to compute Poincare maps. That is, we will plot only certain points when certain events happen. There are several different types of Poincare maps that are possible in *XPPAUT*. If some quantity, say, Q crosses a value, say, v with a positive sign, then we plot all the variables at the time at which this happens. In the present case, we want the quantity `amp` to vanish as that is when the magnitude of z_n is 2. The statements `poivar=amp,poipln=0,poisgn=1` tell *XPPAUT* to check the quantity `amp` hitting 0 with a positive derivative. The statement `poistop=1` means to stop the simulation when this happens.

Start *XPPAUT* with this input file. Turn off the coordinate axes by clicking on `Graphic stuff aXes opts` and setting `X-org` etc to 0. Next, we will do a two-parameter range integration like in the calculation of the bifurcation diagram for the logistic equation. Click on `Initialconds 2 par range` (**I 2**) and fill in the dialog as follows:

`Vary1:` cx	`Reset storage:` N
`Start1:` -1.5	`Use old ic's:` Y
`End1:` .5	`Cycle color:` N
`Vary2:` cy	`Movie:` N
`Start2:` -1	`Crv(1) Array(2):` 2
`End2:` 1	`Steps2:` 200
`Steps:` 200	:

Click on **Ok** to run the simulation. This might take a bit of time; you are running 40000 simulations. You should see the Mandelbrot set appear on your screen. The points in the set itself are empty. The reason for this is that in in the Poincare map option of *XPPAUT* , if the condition never occurs in the allotted time, no points are captured. Thus, a point is plotted at `(cx,cy)` if iterates with that parameter value leave a circle of radius 2 after fewer than 50 iterates.

Since *XPPAUT* knows how many iterates it takes to exit, we can color code the dots accordingly. Look at the first column in the **Data Viewer** . This is the time (interpolated) at which the iterating left the circle of radius 2. Thus, we can code these times by color. Click on `nUmerics Colorcode Another quant` (**U C A**). Accept the default of `T`. Click on **Choose** and choose 2 for `Min` and 12 for `Max`. Click on `Esc` to get to the main menu and redraw the graph by clicking **R** . You should see the familiar, albeit, somewhat crude, colorized Mandelbrot set. To get a nice picture of it, just create a new smaller window which is 200 x 200. Click on `Makewindow Create` and resize it. You should see something like Figure 4.6.

Exercise Make a Julia set animation with the function $f(z) = z^3 + c$ and then make a Mandelbrot set for this function. (Note that for cube roots, you want to randomly choose between three possible roots.) A good range for the cubic Mandelbrot set is c_x between -0.8 and 0.8 and c_y between -1.5 and 1.5. (Hint: $z^3 = x^3 - 3y^2x + i(yx^3 - y^3)$.) Create some art by changing some of the coefficients in the nonlinearities obtained above. Add some quadratic terms, change a few signs, whatever. To speed things up, change the 200 to 100 in the two-parameter range. When you find something good, then do a finer version. You don't have to create a new *XPPAUT* file for this exercise. Just click on `File Edit RHS's` to edit the right hand sides of the equations.

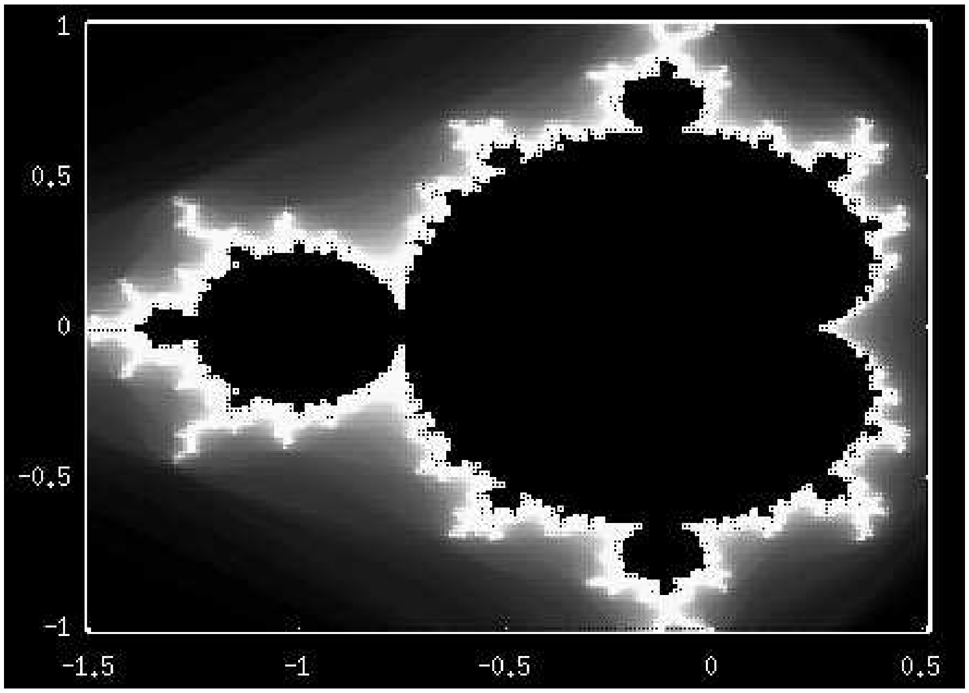


Figure 4.6. *Mandelbrot set.*

4.2.6 Iterated function systems

A variety of interesting fractals can be created by using randomly chosen affine maps in the plane. An affine map is just a transformation of the form

$$x \mapsto Ax + b.$$

If you take a series of such maps, $\{A_n, b_n\}$ along with probabilities, p_n for choosing each one, then this is called an iterated function system (IFS). Starting with some point x is the plane, choose a map, apply it to get a new point, and plot the point. Do this thousands of times and the result will often be an interesting image. Barnsley and others have developed image compression algorithms based on these ideas. However, we will use them to create fractal pictures. As an illustrative example that requires no computer, lets look at a one dimensional example. At each step multiply x by 1/3. Then flip a coin. If it is heads add 2/3, otherwise add 0. From this, it is clear that the middle third of the interval can never be plotted. Similarly, the middle third of the left and right intervals can never be plotted. Thus, the points that are plotted consist of the interval with all the middle thirds removed – the Cantor set. This is a simple example and you can look at it via the following ODE file:

```
# cantor.ode
# plots the Cantor set
x'=x/3+2*heav(ran(1)-.5)/3
y'=.5
@ xlo=0,xhi=1,xp=x,yp=y,ylo=.4,yhi=.6
@ meth=discrete,total=40000,lt=0,maxstor=40001
done
```

Notes. The first equation is exactly the iteration. The term `heav(ran(1)-.5)` is zero if the random number between 0 and 1 is less than .5 and 1 otherwise. The equation for `y` is a dummy; since we want to plot a 1-dimensional set we keep the y-coordinate constant. The statement `lt=0` is to set the line-type to zero which plots single dots.

Run this. Then use the `Window/zoom Zoom in` command to zoom into the left or right halves. Notice you get the same sort of set. Continue zooming in like this. At each step you will see the same set. The Cantor set is self-similar; that is, magnified regions of it look the same as the whole set. Here is another cool trick. Make a histogram for the Cantor set with 100 bins. Click on `Numerics Stochastic Histogram` (**U H H**). Choose `100` for the number of bins, and accept the defaults. Exit the numerics menu by clicking on **Esc** . Next change the line style: click on `Graphic stuff Edit curve` . Choose `0` as the curve to edit, change the linetype to 1, and click on **Ok** . Finally, click on `X vs t` and choose `X` as the plot variable. You will see the histogram for the Cantor set. Go back into the Numerics menu and make another histogram, this time with 1000 bins. and replot it. The histogram has many more dips than before. If you zoom in, you will find the histogram also is fractal.

Now, let's make a two-dimensional iterated function system. We will start with a classic, the Sierpinsky triangle. This fractal is obtained by dividing a right triangle into 4 equal pieces and removing the middle piece. This is repeated in each of the three smaller triangles which remain and so on. The result is like a two-dimensional version of the Cantor set. The iterated function system consist of three maps. Each map consists of first dividing the coordinate in half. Then with probability 1/3, 1/2 is added to the x coordinate, 1/2 is added to the y coordinate, or nothing is added. Here is an ODE file

```
# simple iterated function system
# sierpinsky triangle
par c0=0,c1=.5,c2=0
par d0=0,d1=0,d2=.5
p=flr(ran(1)*3)
x'=.5*x+shift(c0,p)
y'=.5*y+shift(d0,p)
@ meth=discrete,total=20000,maxstor=100000
@ xp=x,yp=y,xlo=0,xhi=1,ylo=0,yhi=1,lt=0
aux pp=p
done
```

Notes. I define 3 values for constants that are added after multiplying by 1/2, `(c0,d0), (c1,d1), (c2,d2)`. Then I flip a "three-sided" coin, `p` which takes values of 0, 1 or 2. The function `shift(c0,p)` will be `c0,c1,c2` according as whether `p` is 0, 1 or 2 respectively.

Run this and you will see the Sierpinsky triangle appear on the screen. Zoom in to see similar versions of it.

We close this section by writing a simple IFS that has 4 general probabilities and 4 general affine maps. It should be clear from this how to generalize to more maps.

```
# ifs4.ode
# general code for ifs with 4 choices
par a1=0,a2=.85,a3=.2,a4=-.15
par b1=0,b2=.04,b3=-.26,b4=.28
par c1=0,c2=-.04,c3=.23,c4=.26
par d1=.16,d2=.85,d3=.22,d4=.24
par e1=0,e2=0,e3=0,e4=0
par f1=0,f2=1.6,f3=1.6,f4=.44
par p1=0.01,p2=0.85,p3=0.07,p4=0.07
s1=p1
s2=s1+p2
s3=s2+p3
z=ran(1)
i1=if(z>s1)then(1)else(0)
i2=if(z>s2)then(2)else(i1)
i=if(z>s3)then(3)else(i2)
x'=shift(a1,i)*x+shift(b1,i)*y+shift(e1,i)
y'=shift(c1,i)*x+shift(d1,i)*y+shift(f1,i)
@ meth=discrete,total=20000,maxstor=100000
@ xp=x,yp=y,xlo=-3,xhi=3,ylo=0,yhi=10,lt=0
done
```

Notes: The first six lines give values for the affine maps:

$$x' = ax + by + e \quad y' = cx + dy + f.$$

The next lines give the four probabilities of each of the maps. Notice that these probabilities are not uniform. The numbers `s1,s2,s3` are the cumulative probabilities. Next, a random number `z` is selected. The next three lines of code determine which map to use based on `z` by testing whether or not `z` is larger than each of the cumulative probabilities. This sequence of code is similar to that used in our implementation of the Gillespie algorithm (see Chapter 5.) Finally, the map is defined by using the shift operator on each of the parameters for the map. Thus, if `i=0` then the first map is used and so on. For a much larger set of parameters, you would likely use look-up tables (see for example the cellular automata example in Chapter 6). If you run this ODE file, you will see the famous fractal fern.

4.3 One-dimensional ordinary differential equations

Many courses in differential equations start off by describing a variety of exactly solvable first order differential equations. However, recent texts such as those by Blanchard et al, Strogatz, and Borelli and Coleman take a more qualitative approach and emphasize graphical and geometric methods for solving differential equations. There are several ways to gain a qualitative view of the behavior of a differential equation. One way is to draw the *phaseline.* Consider the differential equation

$$\frac{dx}{dt} = f(x).$$

To draw the phaseline for this, we plot $f(x)$ versus x. Then we use the sign of $f(x)$ at any point x to determine the direction along the x axis. Fixed points are precisely zeros of $f(x)$. Thus, to draw a phaseline plot, we draw $f(x)$ and then representative trajectories on the x axis. *XPPAUT* can animate this whole process. We will construct an animation file which plots the function $f(x)$ and a moving ball showing the behavior of $x(t)$. The point of this is to see how the value of $f(x)$ determines the speed of the particle. First, we make an ODE file for the scalar system. Then we add the animation file. Here is the ODE file:

```
# phase-line.ode
# one-dimensional autonomous system
f(x)=x*(x-.25)*(1-x)
x'=f(x)
init x=.26
# transformation information for the animator
par x0=-.5,x1=1.5,y0=-1,y1=1
tr(x)=x0+(x/100)*(x1-x0)
ff(x)=(f(tr(x))-y0)/(y1-y0)
@ ylo=-.25,yhi=1.25,total=40,xhi=40
done
```

The file is pretty straightforward. I have defined 4 parameters, `x0,x1,y0,y1` to scale the x axes and the y axes for the animator. The depicted phase space (in the animation) will be the interval $[x_0, x_1]$. The parameters `y0,y1` give the maximum and minimum vertical values in which $f(x)$ is plotted. Since the x-axis is the phase-space, the parameter `y0` should be less than 0 and `y1` should be greater than zero so that $y = 0$ (the x-axis) will be visible. The function `tr(x)` is used by the animator. If you run this file, the usual $x(t)$ versus t will be plotted. The fun begins when we look at the animation. In the animation file, I plot the function $f(x)$ and then animate a little ball along the x-axis. here is the animation file:

```
# animation file for a phase-line
# phase-line.ani
permanent
# this code draws a function ff(x)
line [0..99]*.01;ff([j]);[j+1]*.01;ff([j+1]);$RED;2
```

```
# a straight line along y=0
line 0;-y0/(y1-y0);1;-y0/(y1-y0);$BLACK
transient
# following the dancing ball
fcircle (x-x0)/(x1-x0);-y0/(y1-y0);.02;$BLUE
end
```

Since we have not looked at animation files too much, I will explain this one in detail. (However, you may want to look at the chapter on animation, Chapter 8, first.) Lines that begin with # are comments. The line `permanent` tells the animator that whatever follows is static and is not animated. For example, titles and other things in the animation are usually not dependent on time. The code starting with `line [0..99]` is somewhat mysterious, so we will pick it apart. The general form for the `line` command is

```
line x1;y1;x2;y2;color;thick
```

This will draw a line from `(x1,y1)` to `(x2,y2)` in color `color` with thickness (in pixels) `thick`. The lower left and upper right coordinates of the animation window are (0,0) and (1,1) respectively. The `[0..99]` notation is new; this just tells *XPPAUT* to repeat this line 100 times with a dummy index going from 0 to 99. (See chapter 6 for more details on this notation.) Thus, 100 lines will be drawn. The first coordinate of the first line is 0, the first coordinate of the second line is 1*.01, and so on. The vertical coordinate of the first line is `ff(0)` since `[j]` refers to the index of the current line which is 0. The last line in this 100 line sequence is expanded by *XPPAUT* to be:

```
line 99*.01;ff(99);100*.01;ff(100);$RED;2
```

The function `ff(u)` is defined in the ODE file. Thus, this code simply draws the function $f(x)$. The next line of code tells the animator to draw the horizontal axis scaled to be at $y = 0$. The code `transient` tells *XPPAUT* all that follows should be animated. A filled circle is drawn at the scaled x coordinate of the dynamical system with radius .02 units (that is 2% of the whole animation window) and it is colored blue.

Click on `View axes Toon` (**V T**) to pop up the animation window. Click on `File` in the animation window and load the animation file, `phase-line.ani`. Unless there is an error, you should be ready to run the animation. (If there is an error, then you did not type in the animation file correctly.) Click on the `Go` button in the **animation window** and the graph of f will show up along with a little ball showing the progress of the dynamical system.

Notes.

1. You can edit animation files and load them into *XPPAUT* without having to leave *XPPAUT*.

2. If you click on the little box in the upper right-hand side of the animation window, the animation will run simultaneously with the simulation. This slows down the simulation.

Exercises

1. Try animating the following equation:

$$x' = \sin \pi x + 1.05.$$

   To do this, in the **Main Window** , click on `File Edit Functions` (**F E F**) and change the first function to `sin(pi*x)+1.05` and then click on `Ok` . Then run the ode with $x(0) = -0.5$ as an initial condition. Rescale the phase-variable axes for the animation by setting the parameters, `x0=-.5` and `x1=10.` Then run the animation.

2. Try to create an ODE file and an animation file for a one-dimensional non-autonomous equation. Note that since the function is non-autonomous, we must also redraw the function curve at every frame. This is not the best way to look at nonautonomous 1d systems. The next section suggests a better way. Here are the two files

```
# phase-line2.ode
# one-dimensional nonautonomous system
f(x,t)=x*(x-.25)*(1-x)+a0+a1*sin(w*t)
par a0=0,a1=0,w=0
x'=f(x,t)
init x=.26
# transformation information for the animator
par x0=-.5,x1=1.5,y0=-1,y1=1
tr(x)=x0+(x/100)*(x1-x0)
ff(x,t)=(f(tr(x),t)-y0)/(y1-y0)
@ ylo=-.25,yhi=1.25,total=40,xhi=40
done

# animation file for a phase-line2.ode
# has time-dependence
permanent
# a straight line along y=0
line 0;-y0/(y1-y0);1;-y0/(y1-y0);$BLACK
transient
# this code draws a function ff(x,t)
line [0..99]*.01;ff([j],t);[j+1]*.01;ff([j+1],t);$RED;2
# following the dancing ball
fcircle (x-x0)/(x1-x0);-y0/(y1-y0);.02;$BLUE
end
```

4.3.1 Non-autonomous 1D systems

Non-autonomous systems are somewhat more difficult to study. The exercise above gives you one way to look at them but this is strictly numerical. Consider

$$\frac{dx}{dt} = f(x,t) \tag{4.1}$$

One way to look at such a system is in the $(t,x)-$plane. *Direction fields* are very helpful in understanding the dynamics. At a regular array of (t,x) values in the plane, we draw a little arrow whose direction is that of the vector $(1, f(x,t))$. This tells us the tangent point to the solution to (4.1) through the point (t,x), thus, we can sketch an approximate solution without ever having to actually solve the differential equation. *XPPAUT* has tools for drawing direction fields. However, it only works for two-dimensional dynamical systems. This is no problem. Consider the differential equation:

$$\frac{ds}{dt} = 1 \quad \frac{dx}{dt} = f(x,s).$$

This is identical to (4.1) but is now sitting in two-dimensions. With these considerations, we introduce the general one-dimensional nonautonomous differential equation file:

```
# oned.ode
# generic one-dimensional ODE file
f(x,t)=x*(1-x)
s'=1
x'=f(x,s)
@ xp=s,yp=x,xlo=-5,xhi=15,ylo=-2,yhi=2
done
```

Run *XPPAUT* with this file. The axes are set up so that the time-like variable s is on the horizontal axis and x is on the vertical axis. Click on `Dir.field/flow Scaled dir. field` (**D S**) and change `10` to `15` at the prompt. You will see a direction field appear. (For fun try the nonscaled direction fields, **D D** , and see why I used the scaled ones. In the unscaled fields, the length is proportional to the magnitude of the vector $(1, f)$.) Based on the direction fields, you should have no trouble figuring out the trajectories starting at any point. To check if you are right, click on `Initialconds Mice` (**I I**) and use the mouse to click in a bunch of different points in the window. Click outside the view window to quit inputting initial data.

Hardcopy hint. *XPPAUT* only keeps the latest integration in memory so that if you want hardcopy of all the trajectories you computed along with the direction fields, you won't be able to get it. However, there is a way to keep up to 26 trajectories. Click on `Graphic stuff Freeze On freeze` (**G F O**) to automatically "freeze" the trajectories onto the screen. Now redraw the direction fields, and compute a bunch of trajectories. Finally, keep your work: Click on `Graphic stuff Postscript` and accept the defaults; choose the default file name `oned.ode.ps`; and *XPPAUT* will create a postscript file in your directory which

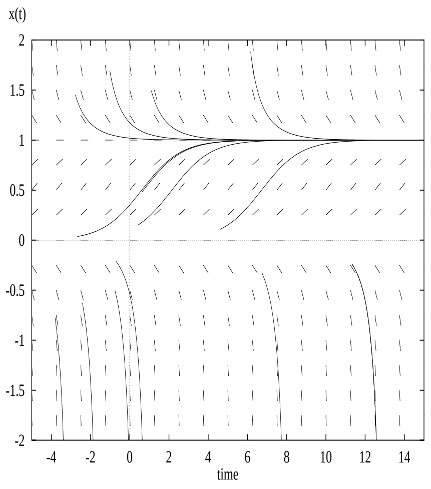


Figure 4.7. *Direction fields and sample trajectories for the equation $x' = x(1-x)$.*

you can plot or view later. Delete the saved trajectories (if you want) by clicking on `Graphic stuff` `Freeze` `Remove all` . Figure 4.7 shows an example.

Exercises Draw direction fields for the following functions $f(x,t)$: (i) $x - t/10$; (ii) $\sin(4x)\cos(t/2)$; (iii) $t - x^2$; (iv) $t/10 - x$; (v) $\sin(xt)$. Try to guess what solutions look like based on the direction fields and verify your guess by computing a number of trajectories. (Hint: To change the function $f(x,t)$ just click on `File` `Edit` `Function` and change $f(x,t)$.

4.4 Planar dynamical systems.

Two-dimensional differential equations play an important role in many undergraduate differential equations courses. This is because of the special topological properties of the plane, in particular, the Jordan Curve Theorem. This constraint implies that the only attractors in the plane are fixed points and limit-cycles. Most textbooks that emphasize the qualitative aspects of differential equations include a chapter or two on planar ODEs. *XPPAUT* has many features that are useful in the analysis of these systems such as the ability to draw direction fields, nullclines, and separatrices for saddle-points. I have used *XPPAUT* with several undergraduate texts in ordinary differential equations. Used intelligently, one can automate the analysis of planar systems and teach some complicated dynamical systems concepts. I will start this chapter with the analysis of two-dimensional linear systems. The general linear system has the form:

$$\begin{aligned} x' &= ax + by \\ y' &= cx + dy \end{aligned}$$

where a, b, c, d are parameters. The behavior of this system is completely determined by the eigenvalues of the matrix

$$A = \begin{bmatrix} a & b \\ c & d \end{bmatrix}.$$

If the eigenvalues are both real and the same sign, the origin is a *node*; if they are real and opposite signs, it is a *saddle*; and if the eigenvalues are complex with nonzero real parts, the origin is a *vortex* or *spiral*. In the cases of nodes and spirals, if the real part of the eigenvalues is negative (positive) the node or spiral is attracting (repelling). There are several degenerate cases. If there is a zero eigenvalue, then the eigenvector corresponding to the zero eigenvalue is a line of fixed points for the system. Finally, if there are pure imaginary eigenvalues, then the origin is called a *center*.

Before starting with the computer, we recall a few facts about linear two-dimensional systems. Define the *trace* to be $T = a + d$, the *determinant* to be $D = ad - bc$, and the discriminant to be $Q = T^2 - 4D$. Then the following holds:

- If $D < 0$ then the origin is a saddle point;
- If $T < 0$, $D > 0$, and $Q > 0$ the origin is a stable node;
- If $T > 0$, $D > 0$, and $Q > 0$ the origin is an unstable node;
- If $T < 0$, $D > 0$, and $Q < 0$ the origin is a stable spiral;
- If $T > 0$, $D > 0$, and $Q < 0$ the origin is an unstable spiral;
- If $T = 0$ and $D > 0$, the origin is a center;
- If $D = 0$, then there is a line of fixed points.

We will illustrate these with a generic two-dimensional linear differential equation with parameters:

```
# twodl.ode
# simple linear planar dynamical system
x'=a*x+b*y
y'=c*x+d*y
par a=1,b=2,c=-3,d=-1
@ xlo=-2,ylo=-2,xhi=2,yhi=2,xp=x,yp=y
@ total=30,nmesh=100
done
```

Note: The file is straightforward; the statement `nmesh=100` tells *XPPAUT* to make a finer mesh for drawing nullclines (which are explained below).

Run this file. Click on `Dir.field/flow Direct field` (**D D**) to draw unscaled direction fields. Since $T = 0$ and $D > 0$, it follows that the origin is a center. Try $d = -2$. This changes the center into a stable spiral. Try $d = -3$. What is the origin? At what value of d does it become a node? At what value of d does it become a line of equilibria? Draw the direction fields and some trajectories when $d = -6$. You should be able to make out the line of fixed points, $y = -x/2$. Change d back to -2. Click on `Dir.field/flow Flow` (**D F**) and change the grid to 5 to draw a flow diagram of the system. You will see a nice vortex. Unfortunately, *XPPAUT* does not save these trajectories, but they do give a nice picture of what is going on.

Hardcopy hint Click on `Kinescope Capture` To grab the screen image. Now click on `Kinescope Save` and choose any name you want for the `base` name, *e.g.* `twodl` and then choose `2:GIF` as the format and a standard GIF file called `twodl_0.gif` will be created. `Kinescope Capture` can be used to capture a sequence of frames and these can be saved in sequence or even animated. Single frames can be viewed with a standard browser or most any graphics software (*e.g.*, `xv` on Linux machines.) On Windows machines, you can capture the window to the clipboard and dump it into a Paint or Word file. (This is done by clicking on the window and pressing the `Alt PrtScrn` key.

Erase the screen and use the mouse to start some initial data at roughly $(2, -2)$. You will get a nice spiral. Let's color code this trajectory according to the absolute velocity, $\sqrt{\dot{x}^2 + \dot{y}^2}$. Click on `nUmerics Colorcode Velocity` (**U C V**) and choose `Optimize`. Escape back to the main menu. Your trajectory is nicely colored with the minimum velocity blue and the maximum red. Let's color the whole phaseplane according to this color scheme. Click on `Dir.field/flow Colorize` and change the grid to 100. You can colorize according to many different quantities (such as energy for a conservative system as we shall see later). While *XPPAUT* does not create a direct postscript version of your picture, you can save the screen image as a GIF file as described above. Individual color-coded trajectories are rendered in the postscript file. Erase the screen (**E**) and go back into the numerics menu and turn off the colorizing (`nUmerics Colorize No color`) and **Esc** back to the main menu.

XPPAUT can tell you the nature of the origin. Click on `Sing pts Go` (**S G**) and answer `Yes` to the dialog box. The *singular point* or *fixed point* or *equilibrium* is computed. If you look in the terminal window from which you started *XPPAUT* , you will see that the eigenvalues are printed and have values of $-1/2 \pm i1.93$. A new window appears which gives a summary description of the stability of the fixed points, the nature of the eigenvalues, and value of the fixed point. From this window, you see that the origin is stable and that it has two complex eigenvalues with negative real parts (as shown by the text `c-=2`). Change the parameter `c` from -3 to 3. Draw the direction fields and a flow diagram. This is a saddle point since $D = -8$ is the determinant. Determine the stability of the origin. Click on `Sing pts Go` and this time answer `No` to the `Print eigenvalues` dialog. Another message box comes up. This asks if you want to draw invariant sets. Answer yes to this. Then press `Enter` four times. You will see 4 trajectories coming out of the origin. They go northeast, northwest, etc. Two are yellow – these are the unstable manifolds of the saddle point and (since this is a linear system) are identical to the eigenvector associated with the positive eigenvalue. The other pair of trajectories are in blue; they are the branches of the stable manifold and are identical to the eigenvectors associated with the negative eigenvalue. When *XPPAUT* drew these, the first two were the unstable manifolds and the second, the stable. Whenever there is a one-dimensional unstable manifold, *XPPAUT* will draw this first. If you erase the screen, you will find that you have lost these manifolds. However, *XPPAUT* keeps the initial data required to recompute them. To draw the *unstable manifolds*, click on `Initialconds sHoot` (**I H**). You will be prompted to choose a number from 1-4. Since the first two trajectories were the unstable manifold, pick either 1 or 2 and one or the other branches will be drawn. To get both of the branches pick the second of the two after freezing the first trajectory. To get the stable manifolds, you must change the direction of integration; that is you must solve the equations backward in time. To solve ODEs backwards in time in *XPPAUT* , click on `nUmerics Dt` and change it to a negative value (e.g. `-0.05`), then exit the numerics menu. Click on `Initial conds sHoot` and choose either 3 or 4 as the stable manifolds are always the last two. Make sure that you change the time step `Dt` back to a positive value if you want to explore the system more. (**Note** as with the flow-fields, you can use the Kinescope to capture the screen with the manifolds as a quick and dirty way of getting a permanent record.)

4.5 Nonlinear systems

Since *XPPAUT* uses numerical methods, it doesn't matter if the system is linear or not. We now turn to the analysis of two-dimensional nonlinear systems:

$$\frac{dx}{dt} = f(x,y) \tag{4.2}$$

$$\frac{dy}{dt} = g(x,y). \tag{4.3}$$

As with linear systems, direction fields can lead to a good qualitative picture of the behavior. Another qualitative method related to the computation of direction field is the use of **nullclines**. Nullclines break the phase plane into regions where $(dx/dt, dy/dy)$ are of constant sign. Thus, they serve to give a sense of the direction of the flow. The x-nullcline (resp. y-nullcline) is the curve defined by $f(x,y) = 0$ (resp. $g(x,y) = 0$). Furthermore, the points where the $x-$ and $y-$nullclines intersect are fixed points. *XPPAUT* easily computes the nullclines; it can even animate them as a function of a parameter.

Suppose that a nonlinear system has a fixed point and that the linearization has no eigenvalues with zero real part. Then, locally, the behavior of the nonlinear system is identical to that of the associated linear system. In particular: (i) the stability is unchanged and (ii) the stable and unstable manifolds of saddle points are tangent to those of the corresponding linearized system. This means that *XPPAUT* can compute both the stability and the stable/unstable manifolds for nonlinear systems.

With these tools, it is easy to put together a complete picture of the dynamics of any two-dimensional system. Let's consider the following example from a textbook:

$$\frac{dx}{dt} = x + y + x^2 - y^2 + 0.1$$

$$\frac{dy}{dy} = y - 2xy + x^2/2 + y^2$$

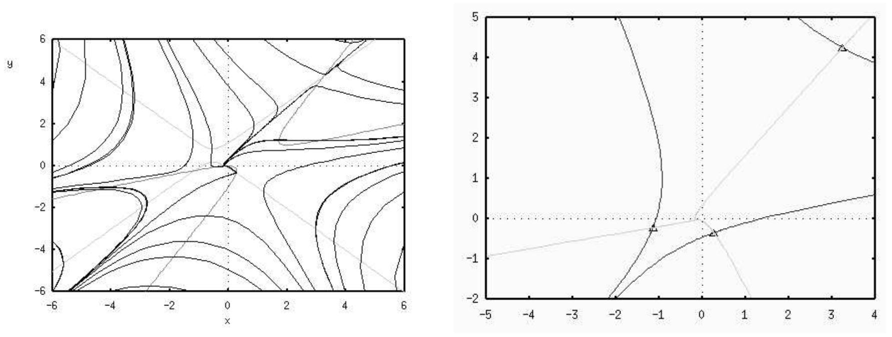


Figure 4.8. *Phase plane showing trajectories and nullclines for the two-dimensional nonlinear example. Below, the stable and unstable manifolds of the fixed points.*

This is not simple to analyze by hand. That's why we use the computer to complete the picture. The first thing to do is to put it into *XPPAUT*. Here is the ODE file `ex2d.ode`

```
# example from Golubitsky
#
x'=x+y+x^2-y^2+0.1
y'=y-2*x*y+x^2/2+y^2
@ xp=x,yp=y
@ xlo=-6,xhi=6,ylo=-6,yhi=6,dt=.02
done
```

I have taken the liberty of setting it up as a phaseplane in a 12x12 window centered at the origin. To get a general picture of what it looks like, plot the direction field: `Dir.field/flow Direct Field` and accept the defaults. Another quick way to get a picture of the behavior is to make a flow picture which draws a bunch of trajectories both forward and backward in time: click on `Dir.field/flow Flow` and accept the default. A nice picture will emerge.

It is not clear how many fixed points there are, but there are certainly more than one. To get a feel for the number of fixed points and where they are, click on `Nullcline New` (**N N**). The x-axis nullcline is red and the y-axis nullcline is green. (**Note:** Sometimes the nullclines look quite choppy. You can make them much smoother by increasing the numerical mesh size – click on `nUmerics Ncline ctl` and change the mesh from 40 to 100. `Escape` back to the main menu, erase the screen and recompute the nullclines.) It looks like there are either 2 or 4 fixed points. We can blow up the region where the fixed points are a bit to resolve this better. Click on the `Window/zoom Zoom` (**W Z**) command and zoom (by clicking the mouse) into an area which contains all the the equilibria. (A good window is (-5,4)X(-2,5)). Click on `Erase` to erase the screen. Redraw the nullclines and the direction fields. (**N N** ; **D Enter Enter**) You can now see clearly that there are 4 fixed points. Now lets assess the nature of the fixed points indicated by the intersections of the nullclines. Click on `Sing. Pts Mouse` and click with your mouse on the upper right fixed point. Answer **No** to the question about eigenvalues and **Yes** to the question about invariant sets. A yellow trajectory (one branch of the unstable manifold) is drawn and a window appears warning you that the trajectory is out of bounds. Press any key and the other branch of the unstable manifold is drawn. Press a key and a branch of the stable manifold (blue) is drawn. Press the `Esc` key to get the other branch. You should see four trajectories for this saddle-point: two in yellow (the unstable manifolds) and two in blue (the stable manifolds). Repeat this at the other 3 fixed points – two more of them are saddles and the fourth one is an unstable vortex. You should see something like Figure 4.8. (In the figure, for clarity, I have split the picture into two parts, showing the flow and nullclines to the left and the stable and unstable manifolds to the right.) Arrows go out of all the yellow/orange trajectories emerging from the saddle points (triangles) and come into the the blue trajectories toward the saddle points. Thus a point starting near one of the blue trajectories will go toward the saddle to which it is connected and then out along a yellow/orange trajectory. Since there are no stable fixed points and no limit cycles in this example, integrating the equations forward in time will almost always result in solutions that go to infinity. Try a few initial conditions to convince yourself of this fact.

We can study a different example; either by writing a new ODE file or editing the right-hand sides of the equation. Let's examine the following equation:

$$\frac{dx}{dt} = x(x(1-x)-y) \quad \frac{dy}{dt} = y(x-.4).$$

Click on `File Edit RHS` to edit the right-hand sides. Enter these two equations (carefully) and then click on **Ok** . Change the window to (-0.25,1.25)X(-0.25,1.25) by using the `Window/Zoom Window` dialog. Draw some direction fields: `Scaled` or `Unscaled`. Recall that scaled direction fields represent only the direction whereas unscaled also change the length as a function of the size of the vector field. I prefer the former, but this is a matter of taste. Try either one. Click on **D S** and use a grid size of 16 (instead of 10) and you will get pure direction fields. Click on **N N** to get the nullclines. You will see a picture like Figure 4.9. There are clearly two fixed points and the third (0,0) is ambiguous – the nullcline algorithm is not perfect. Verify that (0,0) is a saddle point and that the stable and unstable manifolds are the y and x axes respectively. Examine the fixed point $x = 1, y = 0$. Show it is a saddle point as well. In this case, note that one branch of the unstable manifold (in yellow) appears to have a limit cycle as its limit. You will have to click on `Esc` to stop the integration of this trajectory. Finally note that the fixed point $x = 0.4, y = 0.24$ is an unstable vortex. Choose some initial data in the positive quadrant to verify that solutions all tend to a stable limit cycle. Recall that a **limit cycle** is an isolated periodic orbit for a differential equation.

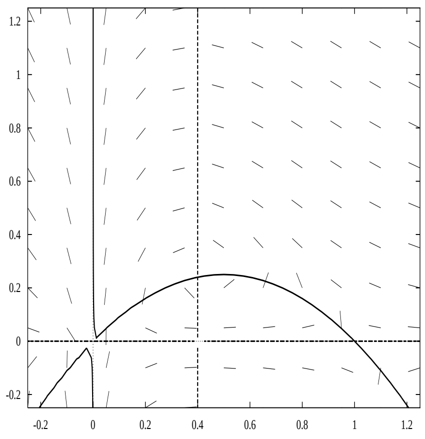


Figure 4.9. *Direction fields and nullclines of a predator prey example.*

4.5.1 Conservative dynamical systems in the plane

The equations for a frictionless particle with mass, m operating under the influence of a potential function $P(x)$ have the following form:

$$\frac{d}{dt}(m\frac{dx}{dt}) = -\frac{\partial P}{\partial x} \equiv -f(x) \tag{4.4}$$

where $f(x)$ is the force and mdx/dt is the momentum. If we write this as a system of equations for the position, x and the velocity, v, we obtain

$$\frac{dx}{dt} = v \quad m\frac{dv}{dt} = -f(x)$$

where we have assumed that the mass is constant. If we multiply (4.4) by dx/dt we can integrate the equation revealing that the total energy (the kinetic plus the potential) is conserved:

$$E = \frac{1}{2}mv^2 + F(x);$$

that is $dE/dt = 0$. In general, we say the system (4.2) has an *integral* if there is a function, $I(x, y)$ such that $dI/dt = 0$ along trajectories. A easy way to test if a system is integrable in two-dimensions is to show that

$$\frac{\partial f}{\partial x} + \frac{\partial g}{\partial y} = 0.$$

In this case an integral is easy to find and I leave it as an exercise. For example the system:

$$x' = xy + y^3 \quad y' = -y^2/2 + x^2(1-x)$$

is integrable and the function:

$$I(x,y) = xy^2/2 + y^4/4 + x^4/4 - x^3/3$$

is constant along solutions. Let's first explore this system and then we will turn to some more standard mechanical systems. Here is the *XPPAUT* file for this system:

```
# integrable.ode
# an integrable system with its integral
x'=x*y+y^3
y'=-y^2/2+x^2*(1-x)
# here is the integral
aux i=x*y^2/2+y^4/4+x^4/4-x^3/3
# here is the log of the integral
aux li=log(x*y^2/2+y^4/4+x^4/4-x^3/3)
@ xp=x,yp=y,xlo=-2,ylo=-2,xhi=2,yhi=2
@ total=100
done
```

NOTE: To improve scaling, I have also computed the log of the integral — taking the log of a quantity can be used to better delineate large orders of magnitude.

I have set up the plot as a phase-plane. Explore this system by setting some initial conditions in the plane. Note that almost every solution that you compute is a periodic. This is a common feature of conservative systems. Check in the **Data Viewer** that the quantity, I remains constant along every trajectory. Turn the freeze option on (`Graphic stuff  Freeze  On freeze`) and then use the mouse to choose a bunch of nice trajectories (`Initial conds  mIce`). Let's colorize the plane according to the log of the integral. *XPPAUT* allows you to colorize trajectories and the phase-plane according to the values of the vector field at each point along the trajectory or in the phaseplane. You choose to color according to the current time along the trajectory, the speed, or some other quantity, like the integral. This other quantity must be a single variable or plottable quantity. Click on `nUmerics` to go to the numerics menu. Then click on `Colorize  Another quantity` and select `li`, the log of the integral as the quantity. Then for the color scaling, select`Choose` . Choose -7.5 as the minimum and 1 as the maximum. `Escape` to the main menu. Click on `Dir.field/flow` and `Colorize` and choose a grid of 100. You should see the plane fill with color. Red dominates as away from the origin, I is large. Notice that near origin, all the colors change rapidly. You should see something like Figure 4.10

Now let's turn to the double well potential, $P(x) = x^4/4 - x^2/2$. The differential equation is

$$m\ddot{x} = x - x^3$$

which we write as a system:

$$\dot{x} = y, \quad \dot{y} = (x - x^3)/m.$$

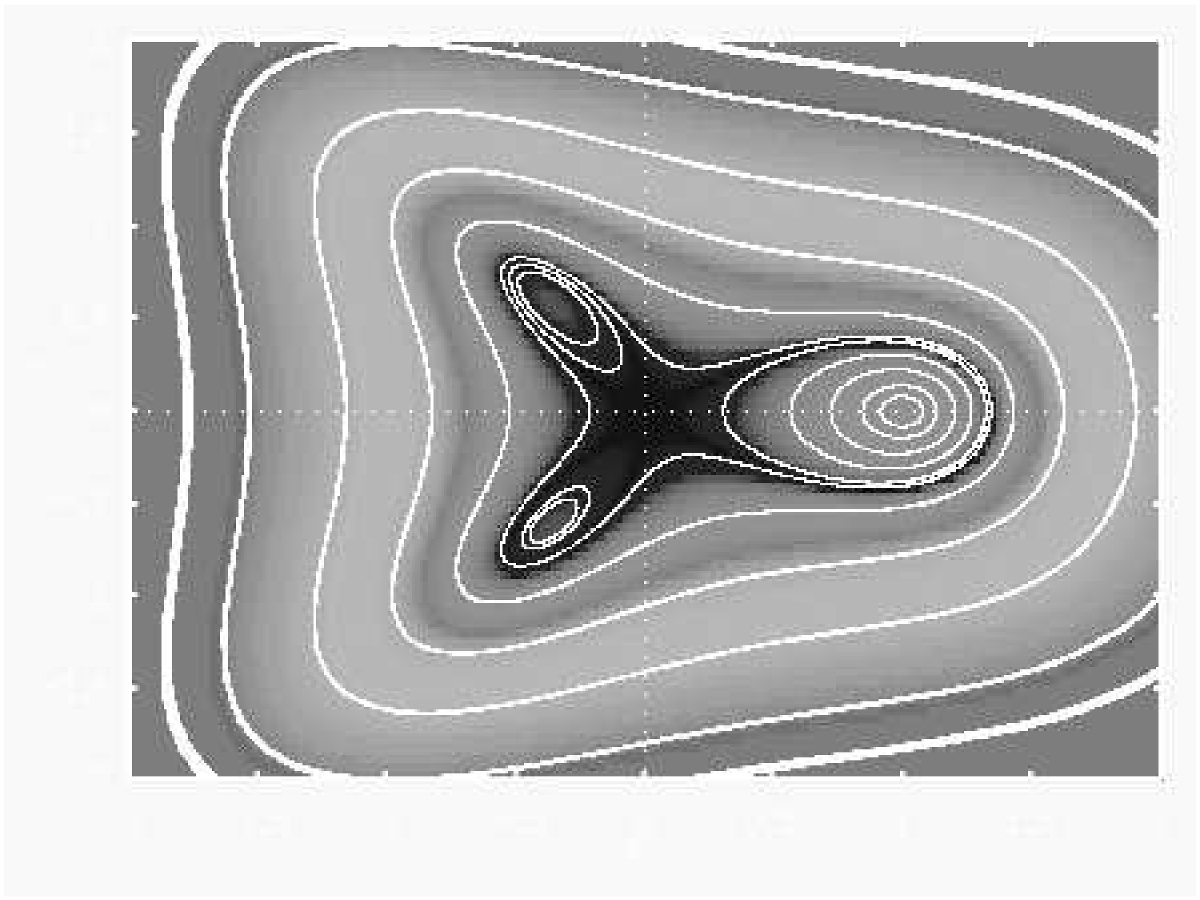


Figure 4.10. *Colorized conservative system: Greyscale indicates log of the integral.*

The total energy is

$$E = my^2/2 + x^4/4 - x^2/2$$

so that you can either write a new ODE file or edit the right-hand sides of the previous one. Here is my ODE file, `double_well.ode`:

```
# double well potential
x'=y
y'=(x-x^3)/m
par m=1
aux e=m*y^2/2+x^4/4-x^2/2
@ xp=x,yp=y,xlo=-1.5,ylo=-1.5,xhi=1.5,yhi=1.5
done
```

Do the following steps:

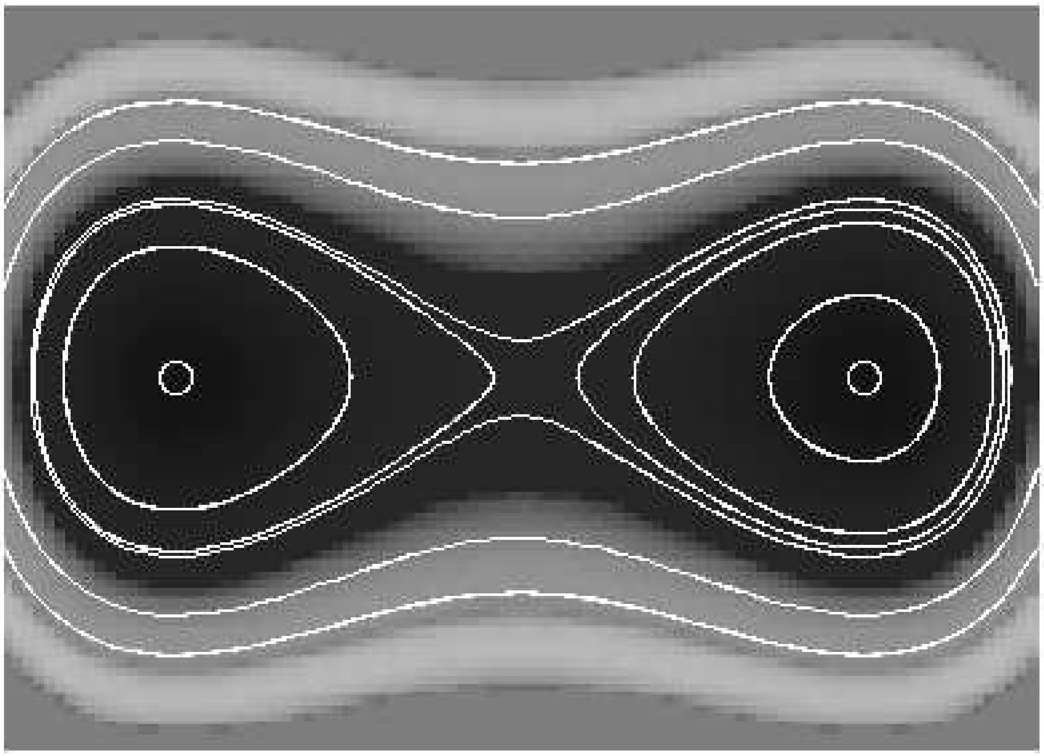


Figure 4.11. *Double well potential*

1. Click on `Graphic stuff` `Freeze` `On freeze` ;
2. Click on `Initial conds` `mIce` and choose 10 or so initial conditions to get some nice trajectories;
3. Click on `nUmerics` `Colorize` `Another quantity` and choose `E` as the quantity and select `Choose` ; set the minimum to -.25 and the maximum to .75; `Escape` to exit to the main menu;
4. Click on `Dir.field/flow` `Colorize` and choose a grid of 100;
5. Redraw by clicking `Restore` .

You should see a nice figure 8 such as in figure 4.11

4.5.2 Homework

Here are some equations to study. Some of these are integrable so you should find the integral. For each of these, draw the nullclines, the direction-fields, the stable and unstable manifolds of the saddle-points.

1. $x' = y(1-x^2)$, $y' = 3 - x^2 - y^2$ in the window (-3,3)X(-3,3). This is an interesting vector field since on one side it looks like it is integrable while on the other side the two fixed points are nodes.

2. $x' = y(1-x^2)$, $y' = x(3-x^2-y^2)$ in the same window.

3. $x' = x + 4x^3 - x^5 - y$, $y' = x$ in the window $-3 < x < 3$ and $-6 < y < 6$. How many limit cycles are there and how many fixed points? (Hint - to find the unstable limit cycle(s), you should integrate backwards in time; click on `nUmerics Dt` and change it to -0.02.)

4. Draw the phase portrait for this system, $x' = y - y^3 - x^3$, $y' = x^3 - x + 3x^2y$. Show that it is integrable. How many fixed points are there and what is their nature. Draw all the connecting trajectories from saddle to saddle.

5. $x' = y(1-y^2)$, $y' = -x(1-2x^2)$. This system can be perturbed to produce a system with 11 limit cycles!.

6. Here is a classic two-neuron neural network:

   $$\tau_1 u_1' = -u_1 + f(w_{11}u_1 + w_{12}u_2 - \theta_1) \quad \tau_2 u_2' = -u_2 + f(w_{21}u_1 + w_{22}u_2 - \theta_2)$$

   where $f(u) = 1/(1+\exp(-u))$. The four numbers w_{jk} are the weights, τ_j are the time constants, and θ_j are the thresholds. The function $f(u)$ is the firing rate as a function of the inputs. This system has an incredible range of different phase-portraits. Type this in as an ODE file.

   a. Using the following values for the parameters: $w_{11} = 12$, $w_{12} = -6$, $w_{21} = 13$, $w_{22} = -2$, $\theta_1 = 3.5$, $\theta_2 = 6$, $\tau_1 = 1$, and $\tau_2 = 1$, draw the nullclines. There are three fixed points. The middle fixed point is a saddle. Draw the unstable and stable manifolds. Note that the two branches of the unstable (yellow) manifolds form a loop which terminates on the left-most fixed point. Now decrease θ_1 to around 3.2 and draw the nullclines. Note that the two lower left fixed points have disappeared. Integrate the equations to see what happens. There is a stable limit cycle that has emerged from the loop formed by the unstable manifolds of the saddle-point (which no longer exists.) This is a classical bifurcation of a large amplitude periodic orbit from a saddle-node on a circle. There is a value of θ_1 in which the saddle point and the stable node coalesce into one point. Try to find this point. (Hint: It is between 3.2 and 3.5.) Change τ_2 to 0.9 and θ_1 back to 3.5. Again look at the stable and unstable manifolds for the saddle-point. Change θ_1 to 3.4 and redo the manifold calculation. Note that the right-branch of the unstable manifold now terminates on a limit-cycle. For θ_1 between 3.4 and 3.5, the right-branch of the unstable manifold, rather than terminating on the stable node, terminates back in the saddle along the stable manifold. This is called a homoclinic loop. For θ_1 slightly smaller than this critical value, a stable limit cycle emerges from this homoclinic. In this parameter regime there are three fixed points (the leftmost stable) and a stable limit cycle. The limit cycle that emerges from the homoclinic orbit is stable because the so called *saddle-quantity*, $\lambda_1 + \lambda_2$ is negative where λ_j are the eigenvalues for the saddle point. (Check this is true!).

b. **More fun with limit cycles.** Verify that there are 3 limit cycles for the neural network model and one fixed point for the following parameters $w_{11} = 8$, $w_{12} = -6$, $w_{21} = 16$, $w_{22} = -2$, $\theta_1 = 0.34$, $\theta_2 = 2.5$, $\tau_1 = 1$, and $\tau_2 = 6$. Determine their stability. (Hint: Unstable limit cycles in planar systems can be found by integrating backwards.) Change $\theta_1 = 0$, $\theta_2 = 3.55$, and $w_{21} = 7$ and show that there is a stable limit cycle, a stable fixed point, and an unstable limit cycle. This bistability occurs for a completely different mechanism from the bistability induced by the homoclinic bifurcation above.

c. Find some parameters such that there are 9 fixed points! (Hint: try w_{11}, w_{22} large and positive (say, around 10), θ_1, θ_2 positive, and w_{12},w_{21} small and negative.

4.6 Three and higher dimensions

XPPAUT is able to create three-dimensional plots of higher dimensional dynamical systems that are occasionally encountered in textbooks. Here, we will examine a few classic examples and then show some fancy projection plots. Consider the Lorenz equations which are an approximation of a large scale atmosphere model. Here are the equations:

$$\begin{aligned}
\frac{dx}{dt} &= s(y - x) \\
\frac{dy}{dt} &= rx - y - xz \\
\frac{dz}{dt} &= -bz + xy.
\end{aligned}$$

The standard parameters are $s = 10, r = 27, b = 8/3$. The *XPPAUT* file that I show below is rather complicated since I will do some fancy graphics tricks. Here is the idea: I will draw a three-dimensional plot and then on one or more of the coordinate planes, plot the corresponding two-dimensional projections, as in Figure 4.1. We will also explore Poincare maps, Fourier spectra, and Liapunov exponents with this example. Here is the ODE file, `lorenz.ode`

```
# the lorenz equations
# with some fancy graphics options
x'=s*(-x+y)
y'=r*x-y-x*z
z'=-b*z+x*y
par r=27,s=10,b=2.66
init x=-7.5,y=-3.6,z=30
# now add some projection planes for 2D plots
aux x2=xplane
aux y2=yplane
aux z2=zplane
par xplane=-35,yplane=60,zplane=-10
# set up the numerics
@ total=50,dt=.02
# set up 3D plot
@ xplot=x,yplot=y,zplot=z,axes=3d
# tell XPP there are 4 plots altogether
@ nplot=4
# here are the 3 projections
@ xp2=x,yp2=y,zp2=z2
@ xp3=x,yp3=y2,zp3=z
@ xp4=x2,yp4=y,zp4=z
# set up the 3D window
@ xmin=-40,xmax=18,ymin=-24,ymax=64,zmin=-12,zmax=45
@ xlo=-1.4,ylo=-1.7,xhi=1.7,yhi=1.7
# and rotate the plot a bit so it looks nice
@ theta=35
done
```

Notes: As I mentioned, this looks quite complicated for a simple ODE. The first few lines are the actual equations. Then, I add three auxiliary variables which are just constants. So if I plot `(x,y,z2)` for example in three-dimensions, the z-coordinate is held constant. This results in a projection in on the (x, y) plane. The choice of planes was done after some experimentation in order to give the best view. The rest of the file is simply setting *XPPAUT* 's plot parameters (all of which could be done within the program). The comments should make it clear exactly what I am doing. The parameters `phi, theta` control the view angle for three-dimensional graphs.

Run *XPPAUT* with this file and integrate the equations. You should see something like Figure 4.12 There are three two-dimensional projections in various Fall colors and the big white Lorenz attractor in the middle. The axes have labels in the figure. To label axes, click on `View axes 3D` and fill in the last three dialog box entries. We can rotate this graph around the vertical axis in the form of a little movie to get a better view of the attractor. Click on `3D params` and fill in the right-half of the dialog box as follows:

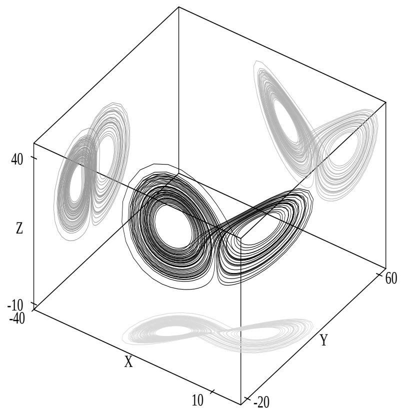


Figure 4.12. *The Lorenz attractor*

`Movie(Y/N):`	Y
`Vary (theta/phi):`	theta
`Startangle:`	35
`Increment:`	15
`Number increments:`	23

This tells *XPPAUT* that to make a series of 23 screen captures in which the rotation angle `theta` is varied by 15 degrees at a time starting with 35 degrees. These screen shots can be played back smoothly like a movie using the `Kinescope` command. Click on `Ok` after you have filled in the dialog box. (Make sure that the window in which the graph is drawn is not obscured by any window other than at most the dialog box. Otherwise the capture will not work properly.) A series of pictures will be drawn on the screen. These are captured in a buffer and can be played back, either one frame at a time, or in smooth succession using the `Kinescope` command. Click on `Kinescope Autoplay` to smoothly run through all the captured screens. You will be prompted for the number of loops you would like: choose 3; and the amount of time between snapshots in milliseconds: 50-100 are reasonable values. You will then see the three-dimensional plot smoothly rotated and this is repeated 3 times. If you want to manually step through the screen shots, choose `Kinescope Playback` . Clicking any mouse button advances to the next frame. Clicking **Esc** gets you out of the loop. You can save these as an animated GIF file also by using the `Kinescope Make Anigif` command. The file is always saved as `anim.gif`.

4.6.1 Poincare maps, FFTs, and chaos

The Lorenz equation is famous as it was one of the first numerical examples of chaotic behavior. There are many ways to characterize chaos in a deterministic dynamical system. Two of the most common characteristics are the existence of infinitely many unstable periodic orbits. There is a classic mathematical theorem due to Sarkovskii who showed that for discrete dynamical systems, if there is period 3 orbit then there are infinitely many periodic orbits. This was later reproved by Li and Yorke in a classic paper "Period three implies chaos." Recall that we found all the period 3 orbits in the logistic equation in section 4.2.2. Another way (and better known) to characterize chaos is through sensitive dependence on initial conditions. That is, if one starts with two initial conditions close to each other, the solutions will diverge from each other at an exponential rate.

Sensitive dependence. Delete the graphs other than the original plot of the Lorenz equation by clicking `Graphic stuff Remove all` (**G R**). Then plot X as a function of time in your window by clicking **X** and choosing `x` at the prompt. Freeze this curve (**G F F**) and choose color 1 (red). Now in the **Initial Data Window**, change the initial value of `z` from `30` to `30.001` and then click on the **Go** button. The trajectories are pretty close up to about $t = 10$ where after they diverge. This gives a ballpark estimate of the rate of divergence. The exact rate of divergence is governed by the maximal Liapunov exponent. This is a numerically determined quantity which is found by integrating a linear differential equation obtained by linearizing the original equation around the trajectory. By definition, the divergence, $d(t)$ away from the trajectory is approximately:

$$d(t) = d(0)\exp(\lambda t)$$

where λ is the maximal exponent. Taking $t = 10$, $d(0) = 0.001$ and $d(10) = 1$ we get that $\lambda \approx \ln(1000)/10 = 0.69$ which is close the actual published value of about 0.9. You can do better than this by clicking on `nUmerics Stochastic Liapunov` (**U S L**), choosing `0` to not get a range of values, and then clicking on **Esc** to get back to the main menu after the exponent is presented. The number you get here is also a little low. You can improve the estimate by integrating away all the transients and integrating for a long time. If you use the `Initialconds Last` (**I L**) and extend the time of integration to 99, you will get a value closer to 0.9. The main point is that the maximal Liapunov exponent is **positive** which implies that there is divergence of nearby trajectories. (**Note.** *XPPAUT* computes the maximal Liapunov exponent along a computed trajectory by linearizing about each point in the trajectory, advancing one time step using a normalized vector, computing the expansion, and summing the log of the expansion. The average of this over the trajectory is a rough approximation of the maximal Liapunov exponent.)

We can see this sensitivity in a rather dramatic fashion by solving the Lorenz equations over 50 different initial conditions that are close to each other and then looking at solutions in a two-dimensional projection. We will use the animator for this and create an ODE file with 50 independent versions of the Lorenz equation. The file is called `lorenz50.ode` and here it is:

```
# 50 lorenz equations
x[1..50]'=s*(y[j]-x[j])
y[1..50]'=r*x[j]-y[j]-x[j]*z[j]
```

```
z[1..50]'=-b*z[j]+x[j]*y[j]
par r=27,s=10,b=2.66
init x[1..50]=-7.5,y[j]=-3.6
init z[1..50]=30.00[j]1
@ total=50,dt=.02
done
```

Notes. I have used an idiosyncratic aspect of the *XPPAUT* file reader to initialize `z[j]`. The expression `[j]` is expanded to the number value of `j` so that if `j=15`, the expression `30.00[j]1` is expanded to `30.00151`. Thus, the initial data are all close but differ slightly! The trailing "1" is included to distinguish `30.0041` (`j=4`) from `30.00401` (`j=40`).

Run this program and integrate the equations (**I G**). Now, click on `View axes Toon` to bring up the animation window. The animation file for this illustration is called `lorenz50.ani` and has the following form:

```
# animation for lorenz50.ode
fcircle (x[1..50]+20)/40;z[j]/50;.02;[j]/50
done
```

What this does is plot little colored balls in the (x, z) plane where the coordinates have been suitably scaled. (If you want to know exactly what is going on in this animation file, see the section on animation.) In the **animation window** , click on `Go` to start the animation. You will see a single little ball that becomes a comet as the individual trajectories diverge and eventually, the attractor is filled out by the 50 little balls.

Exit from the file `lorenz50.ode` and now we will get back to work on the Lorenz equation. If you closed the original Lorenz file, you should start it up again and remove the projection graphs. Integrate the equation and plot x against time. Now, we will look at the power spectrum. Click on `nUmerics Stochastic Power` and choose `X` at the prompt. Then click on **Esc** to get back to the main menu. The power is in the first second data column (`x`) and the mode number is in the first data column (`t`) so click on `Xvst` and choose `X` to plot the spectrum. You may want to zoom into the lowest 200 or so modes. Notice the spectrum has no real peaks, indicative of chaos.

4.6.2 Poincare maps

As a final look at the Lorenz attractor, we will compute a Poincare map. For our purposes, a Poincare map consists of a discrete set of values picked whenever one of the variables passes through some prescribed value. In Lorenz's seminal paper, he chose to plot successive maxima of the variable z. We will do this now and will see that the map turns out to be essentially one-dimensional. Note that since we are fixing one variable in a three-dimensional system, we expect to see a two-dimensional map. This computation may take a few seconds as we have to make a small time step (for accuracy in finding the maximum) and also, we should integrate for a long time. Click on `nUmerics` to get the numerics menu. Click on `Total` and change the total to 200. Click on `Dt` and change the time step to 0.01. Now click on `Poincare map` and `Max/min` to select maxima or minima of a variable. Fill in the resulting dialog box as follows:

Variable: Z
Section: 0
Direction(+1,-1,0): 1
Stop on sect (y/n): N

and click on `Ok` . What you have done is tell *XPPAUT* that you want to plot all the variables every time Z has a local maximum (the +1 direction is for local maxima, -1 for local minima, 0 for both) and to not stop when this maximum is crossed. Escape to the main menu `Esc` . Now, integrate the equations. This could take several seconds. Now, we have a series of values (t_n, x_n, y_n, z_n) that are the values of the variables at every time that $z(t)$ has a local maximum. Lorenz plotted z_{n+1} against z_n to obtain a nice one-dimensional map. *XPPAUT* allows you to do this very easily. We will first set up a view with z plotted against itself. Click on `Viewaxes 2D` and fill in the resulting dialog box as follows:

X-axis: Z	Xmax: 45
Y-axis: Z	Ymax: 45
Xmin: 30	Xlabel:
Ymin: 30	Ylabel:

and click on `Ok` . You will see a diagonal line from one corner to the other. Click on `Graphics Edit` and choose `0`. Change the `Linetype` to `-1` and click `Ok` . The diagonal line becomes a series of points. Now, how can we see the map. First, lets freeze this diagonal line of points since with maps, it is always nice to plot the line $y = x$. Click on `Graphics Freeze Freeze` and then click on `Ok` . The diagonal dots become a solid line. *Finally* we can plot the curve z_{n+1} against z_n. *XPPAUT* has a command called `rUelle` named after the famous mathematician, David Ruelle, who proved that you could reconstruct the dynamics of arbitrary systems from the time series of one variable, $x(t)$ by looking the points $(x(t), x(t-\tau_1), \ldots, x(t-\tau_n))$. Thus, we will look at (z_{n-1}, z_n). We thus want to shift the values on the x axis up by one. Click on `nUmerics rUelle` (**U U**) and change the `X-axis shift` to `1`. `Escape` to exit the numerics menu and click on `Restore` to restore the plot. You should see a cusp-like curve that is almost one-dimensional. It intersects the diagonal at around $z = 38.5$ and this corresponds to a periodic point. If you are reasonably good at guessing formulas for graphs from their shape, you might try to approximate this curve by some function of z. Here is my guess:

$$f_{lorenz}(z) = 27 + 18\exp(-|z - 37.3|/5). \tag{4.5}$$

In any case, we see that the Lorenz equations can be well-approximated by a one-dimensional map. How do we find other periodic points? If we change the `X-axis shift` to `2`, for example, we will be plotting z_{n-2} against z_n so that intersections

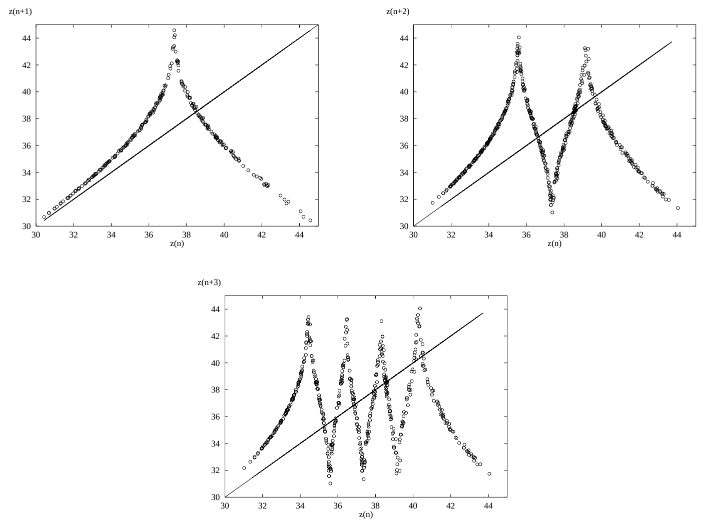


Figure 4.13. *Poincare map for the Lorenz attractor looking at successive values of the maximum of z. Period 1,2, and 3 points are illustrated.*

with the diagonal are period 2 and period 1 points. Try this and you will find two period 2 points (one period 2 orbit). Now for the final blow, lets find the period 3 points. Change the `X-axis shift` to 3 in the Ruelle plot and redraw the graph. You should see 6 intersections besides the trivial one corresponding to two period 3 orbits. Figure 4.13 shows the numerically computed map and the second and third iterates. In chapter 9 (section, 9.6.4) we present a different way to make Poincare maps which can be significantly faster.

Through three lines of evidence: (i) sensitivity to initial conditions, (ii) complex spectrum, (iii) and period 3 orbits, you can be reasonably sure that the Lorenz equations are chaotic. In fact, there are only a few rigorous results on this system.

4.6.3 Homework

1. Explore the simple map f_{lorenz} (equation (4.5) and verify that the maximal Liapunov exponent is about 0.64 so that it is "not as chaotic" as the real Lorenz system. Find values for the period 1, 2 and 3 orbits. Integrate this for a long time and try to show that there are 6 different period 5 orbits.

2. Analyze the Rossler equation:

$$\begin{aligned} x' &= -y - z \\ y' &= x + ay \\ z' &= bx - cz + xz \end{aligned}$$

where $a = 0.36, b = 0.4, c = 4.5$ with initial data $(x, y, z) = (0, -4.3, -0.054)$. First integrate it for a total of 200 with a timestep of 0.1. Look at the time series and various phase-space projections. Look at it in three-dimensions. (Note that the `Window/zoom Fit` is invaluable since this will compute the scales for you!) Compute the Liapunov exponent. Compute the power spectrum. You should see a big peak and some side peaks, but there are many other frequencies. However, it is not nearly as chaotic as the Lorenz system as evidenced by the dominance of certain frequencies and the Liapunov exponent that is close to 0.

Compute a Poincare map for this attractor as follows. Instead of selecting `Max/min` choose `Section` . Choose `X` as the variable, `0` as the section, and +1 as the direction – a point will be stored each time that x crosses 0 with a positive derivative. Integrate for a total of 2000. Plot y_{n-1} against y_n using the Ruelle plot. The resulting map is essentially one-dimensional and looks like an upside down logistic map. Find period 2 and period 3 points. For fun, simulate the map:

$$y_{n+1} = -6 + 0.65(y_n + 3.5)^2$$

with $y_0 = -5$ which is a reasonable fit to the true map. Find the period 1,2 and 3 points for this.

3. Simulate the Chua circuit whose *XPPAUT* file is given below

```
# chuas scroll chaos
x'=a*(y-if(x>=1)then(m1*x+(m0-m1))\
                else(if((x+1)>0)then(m0*x)else(m1*x-(m0-m1))))
y'=x-y+z
z'=-b*y
par a=9,b=14.28
par m0=-0.1428,m1=0.2856
init x=.1,y=.1,z=.1
@ total=200
done
```

Plot it in 3-dimensions and see why it is called scroll chaos. Compute the spectrum and the Liapunov exponent. Take a few different Poincare sections; see if you can find one which reveals a one-dimensional map lurking beneath.

4. Chaos occurs in two-dimensional systems as well if they are periodically forced. The classic example is the forced Duffing equation which we will write as

$$\ddot{x} + x(x^2 - 1) + f\ddot{x} = a \cos t$$

We can rewrite this as a system

$$x' = v \qquad v' = -fv + a\cos(t) - x(x^2 - 1).$$

Write an ODE file. Set $a = .3$ and $f = .25$. Integrate this for a total of 200 and look at it in the (x, v) plane. Window the plane $[-1.5, 1.5] \times [-1, 1]$.

Now, we will take a Poincare map with respect to t, plotting a point every time t hits multiples of 2π. If you choose your section variable to be t, then *XPPAUT* assumes it is periodic and plots a point every time t is a multiple of the section value. Thus, to get a map, choose the Poincare map option, choose `Section` , choose `T` as the variable and type in `6.283185307` for the section. Change the total time to integrate to 10000. Then in the main menu, change the linetype to 0 so dots are pointed (`Graphics stuff Edit` and choose 0 as the graph to edit. Finally, turn off the axes by clicking `Graphics stuff aXes opt` and set change both `X-org` and `Y-org` from 1 to zero. Now run the simulation. You will see a beautiful folded map. This structure arises due to something called a Smale horseshoe that can be found in this system. The horseshoe was one of the first examples of rigorously proven chaos in a dynamical system. If you let the friction f and the forcing a be small in the forced Duffing equation, it is possible to rigorously prove that there is a Smale horseshoe for the system by simply finding some zeros of a certain integral! This is one of the few general methods we have of rigorously proving chaos in a dynamical system.

A simple map that has a similar folded structure is the Henon map:

$$x_{n+1} = 1 - ax_n^2 + y_n \quad y_{n+1} = jx_n$$

which is a two-dimensional map. Here is the ODE file, `henon.ode`:

```
# the henon map
x'=1-a*x^2+y
y'=j*x
par a=1.4,j=.3
init x=.316,y=.206
@ xp=x,yp=y,xlo=-1.3,xhi=1.3,ylo=-.4,yhi=.4
@ total=20000,meth=discrete,lt=0,maxstor=40000
done
```

It is set up to iterate 20000 times. I have to tell *XPPAUT* to increase the storage, `maxstor=40000`, so that we can keep all the points. I have also told *XPPAUT* to use linetype 0. Try this and then zoom into the folded parts. See that it is like pastry all folded up tighter and tighter.

5. The Field-Worfolk equations arise as a four-dimensional normal form for certain symmetric bifurcations. The equations are

$$\begin{aligned} x' &= (\lambda + ar^2 + by^2 + cz^2 + dw^2)x + eyzw \\ y' &= (\lambda + ar^2 + bz^2 + cw^2 + dx^2)y - exzw \\ z' &= (\lambda + ar^2 + bw^2 + cx^2 + dy^2)z + exyw \\ w' &= (\lambda + ar^2 + bx^2 + cy^2 + dz^2)w - exyz \end{aligned}$$

where $r^2 = x^2 + y^2 + z^2 + w^2$ and a, b, c, d, e are parameters. The parameter λ is the bifurcation parameter. The equations are notable because for some values of $a - e$ as λ crosses 0, there is bifurcation to instant chaos. Try $a = -1, b = 0.1$, $c = -.05$, $d = .015$, $e = .55$ and vary λ. Use initial conditions $x = y = w = 0.1$ and $z = 0.2$ You should integrate these for a long time. Here is a possible *XPPAUT* file:

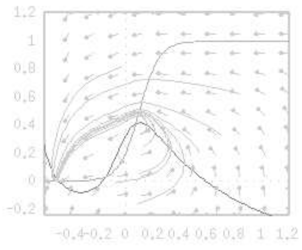
```
# field-worfolk equations
x' = (l+a*r+b*y^2+c*z^2+d*w^2)*x+e*y*z*w
y' = (l+a*r+b*z^2+c*w^2+d*x^2)*y-e*x*z*w
z' = (l+a*r+b*w^2+c*x^2+d*y^2)*z+e*x*y*w
w' = (l+a*r+b*x^2+c*y^2+d*z^2)*w-e*x*y*z
r=x^2+y^2+z^2+w^2
par l=1,a=-1,b=.1,c=-.05,d=-.1,e=1
set fw1 {l=1,a=-1,b=.1,c=-.05,d=-.1,e=1}
set fw2 {l=1,a=-1,b=.1,c=-.05,d=.015,e=.55}
set fw3 {l=1,a=-1,b=.1,c=-.085,d=.005,e=.572837}
init x=.1,y=.1,z=.2,w=.1
@ maxstor=20000,dt=.5,meth=qualrk,tol=1e-5,total=4000
done
```

Try the second set of parameters, `fw2`.

6. As a final example, consider the simplest equation (that I know of) that has chaotic behavior and approaches it through a series of period-doubling bifurcations:

$$\begin{aligned} x' &= y \\ y' &= z \\ z' &= -cx - by - az + x^2 \end{aligned}$$

Choose $a = 1, b = 2$ and let c vary from 3 to 3.5. This system is very similar to the Rossler attractor.



Chapter 5

More advanced differential equations

In this chapter and the next, we will go beyond the usual simple ordinary differential equations that arise in simple applications. Many physical problems do not have a local dependence on time; rather, the state of the system depends on either the complete history of what has already happened or on the state some finite time earlier. *XPPAUT* has many tools to let the user solve and analyze such functional equations.

Many biological systems are not deterministic but instead have stochastic components. These can take many forms such as thermal noise (Brownian motion) and random switching of states (e.g., the random opening and closing of channels). *XPPAUT* has a variety of tools that let you study and simulate these random systems.

Finally, in chemical and mechanical systems, there are often algebraic constraints which relate variables. This leads to so-called differential-algebraic systems (DAE's). *XPPAUT* is able to solve a restricted class of such models. This chapter shows you how to write ODE files for these more complicated model systems.

5.1 Functional Equations

A large class of interesting dynamics are described by functional equations such as delay equations and Volterra type integral equations. *XPPAUT* allows you to numerically solve delay equations as well as solve a variety of Volterra equations.

5.1.1 Delay equations

We will start with some delay equations. Consider the delayed logistic equation:

$$\frac{dx}{dt} = rx(t)(1 - x(t - \tau))$$

where $\tau \geq 0$ is a delay. The way to write this in *XPPAUT* is to use the `delay` function. `delay(x,r)` returns $x(t - r)$ where r is a non-negative number. Here is the ODE file:

```
# delx.ode
# delayed logistic equation
x' = r *x *(1-delay(x,tau))
par r=2,tau=0
init x=.5
aux dlx=delay(x,tau)
@ total=50,delay=10,yhi=6,xhi=50
done
```

NOTES. I have added an extra plottable quantity `dlx` which represents the delayed value of `x`. I have also added the line `@ total=50,delay=10,yhi=6,xhi=50`. This tells *XPPAUT* to integrate for a total of 50 time units, the maximum that the delay will ever be is 10, set the plot maximum for the y axis to 6, and set the maximum for the x axis to 50. *XPPAUT* by default sets the maximum delay to 0. You can also set this maximum from within the program using the `nUmerics dElay` command (**U E**) to set the delay.

Start this up and integrate it. Then change the parameter τ to 1. Integrate it again; note the oscillation. Try a smaller value of τ. Find the critical value of τ above which the constant steady state is unstable. You can do this automatically in *XPPAUT*. *XPPAUT* can track the stability of fixed points of delay equations and will attempt to find a positive eigenvalue if it exists. Try this: set the initial condition for x to be 1 and integrate so that all you see is the horizontal line $x = 1$. Click on `Sing. pts Range` and fill in the resulting dialog box as follows:

`Range over:` tau
`Steps:` 20
`Start:` 0
`End:` 2
`Shoot (Y/N):` N
`Stability col:` 2
`Movie (Y/N):` N

and then click on **Ok** . A bunch of numbers should scroll by in the console window. If you look in the **Data Viewer** you will see the numbers 0 to 2 in increments of 0.1 in the `Time` column and some other numbers in the `X` column. The first column contains the values of `tau` through which you ranged and the second column contains the real part of one of the infinitely many eigenvalues for the linearized equation. (There are infinitely many such eigenvalues since the characteristic equation is transcendental and involves exponentials.) Click on `X vs t` and choose `X` to plot. You will see that the real part of the eigenvalue crosses the axis at about 0.8. Thus for $\tau > 0.8$ the fixed point is unstable and there is probably a Hopf bifurcation. For $\tau < 0.8$ the rest state is stable. Try setting `tau=.75` and integrating with $x(0) = .8$ and then doing the same for `tau=.85`. In the former case, you should see a damped (not damped much!) oscillation while in the latter, the solution goes to a periodic orbit. If you click on `Sing pt Go` , you should see the little equilibrium window indicate the fixed point is unstable. If you look at the console window, you will find that the unstable root is $0.066 \pm 1.88i$. The theory of Hopf bifurcations

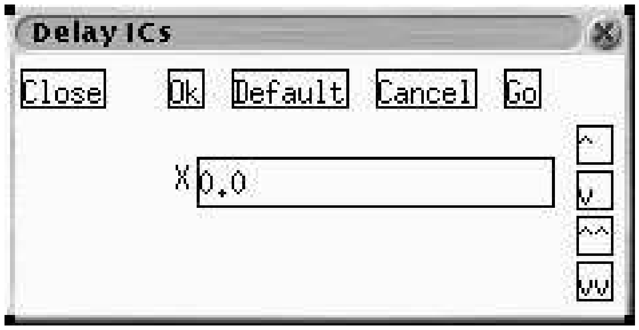


Figure 5.1. *The initial condition window for delay equations.*

implies that the periodic orbit that arises from the destabilization of the fixed point has a period of roughly $2\pi/1.88 \approx 3.34$, where the denominator is the imaginary part of the unstable eigenvalue. Verify that the actual period is around 3.6 which is pretty close!

Unlike ordinary differential equations, delay equations require an entire interval of initial data. *XPPAUT* lets you input this by editing the **Delay ICs window** shown in Figure 5.1. To get this window, click on the button labeled `Delay` on the top line of the **Main Window**. Click in any variable and put a formula of `t` and then click on `Ok` or `Go` . This will initialize the variable for t between 0 and $-r$ where r is the maximal delay. You can also set initial conditions in the ODE file by adding a line of the form:

```
x(0)=f(t)
```

where `x` is the variable.

The delays do not need to be constant; they simply must be non-negative and smaller that the maximum delay. Here is a cool example of a problem due to Ken Cooke. The equation is:

$$\frac{dx}{dt} = rx(t)(1 - x([t]))$$

where $[t]$ is the integer part of t. This is half-way between an ordinary differential equation and a map. In fact, integrating it from $t = [t]$ to $t = [t] + 1$ produces a map that can be easily analyzed. Nevertheless, with a little thought, we can write an ODE file for this model. We note that $[t] = t - (t - [t])$, thus, $x([t]) = x(t - r)$ where the delay, $r = t - [t]$. *XPPAUT* has a function called `flr(t)` which returns the largest integer less than `t`. This will do the trick:

```
# cook.ode
# a delay equation due to Ken Cooke
# x' = r x (1- x([t]))
# where [t]=integer part of t
#
```

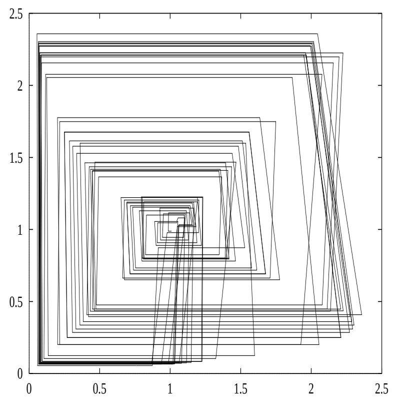


Figure 5.2. *Solutions to Ken Cooke's delay equation.*

```
frac(t)=t-flr(t)
x'=r*x*(1-delay(x,frac(t)))
aux xd=delay(x,frac(t))
par r=2.2
init x=.8
@ delay=2,total=100
done
```

I have added the delayed variable as an extra plottable quantity. The user-defined function `frac(t)` takes the fractional part of `t`. Since $[t]$ is constant over $(n, n+1)$ where n is an integer, this means that $x([t])$ is also constant, say, $x(n)$. Thus $x' = rx(1 - x(n))$ in the interval and you can integrate this equation to obtain a map for $x_n = x(n)$:

$$x(n+1) = x(n)e^{r(1-x(n))}.$$

This unimodal map goes through the usual period doubling cascade to chaos. Thus, the delay equation is also chaotic. Try the delay equation for values of r between 2 and 3. Also, write an ODE file for the map and iterate it for different values of r. Figure 5.2 shows a solution to this equation when $r = 3$ in the $x(t), x([t])$ phase-plane.

5.1.2 Integral equations

There are many physical problems in which the natural setting is not a differential equation but rather an integral equation. *XPPAUT* can solve a variety of Volterra integral equations. The most general forms that can be solved by *XPPAUT* are

$$u(t) = f(t) + \int_0^t K(t, s, u(s))\, ds$$

$$\frac{du}{dt} = f(t) + \int_0^t K(t, s, u(s))\, ds$$

where $f(t)$ is an arbitrary time-dependent function (including dependence on u or other variables). In general, if there is a way by differentiating to convert the integral equation to a differential equation, then *XPPAUT* will be much more efficient than it is solving the integral equation. The reason is basically that to solve an integral equation, you have to integrate over an increasingly longer interval. There are a number of ways to speed *XPPAUT* up. For example, if the integral equation involves a convolution, then, you can tell *XPPAUT* this fact and the integration goes much more quickly. Let's solve an example that is interesting due to the bifurcations that it undergoes as a parameter is varied. The equation is:

$$u(t) = f(t) + r\int_0^t e^{(t-s-\tau)^2} u(s)(1-u(s))\, ds.$$

The function $f(t) = ae^{-bt}$ is transient and serves to force u away from 0, which is clearly a solution in absence of $f(t)$. This is a useful trick to know if you are only interested in long-time behavior. This transient pushes u away from zero. Here is the ODE file

```
# chao_int.ode
# a chaotic integral equation
# I take advantage of the convolution to make a lookup table
x(t)=c*exp(-b*t)+r*int{exp(-(t-tau)^2)#x*(1-x)}
par tau=3,b=8,c=.1,r=2.4
par d1=1.8,d2=3.6
# I add two more variables so we can make a cool 3D plot
aux y=delay(x,d1)
aux z=delay(x,d2)
@ delay=4,total=300,trans=100,dt=0.05
done
```

Here are some comments on the file:

- The construction `int{f(t)# g(t)}` performs the convolution, $J(t)$:

  $$J(t) = \int_0^t f(t-s)g(s)\, ds$$

  and creates a lookup table for $f(t)$ so that it is evaluated very rapidly.

- I have added two additional variables `y,z` so that I can plot (x, y, z) in three dimensions. (y, z) are just delayed versions of $x(t)$. We could have done this inside *XPPAUT* with the `rUelle` plot option under the `nUmerics` menu which allows you to plot delayed versions of the variables on different axes.

- I also skip the initial 100 time units of transients with the statement `trans=100`.

If you run this, you will notice a big slowdown as time goes on. The reason for this is that at t progresses, the domain of the integral gets larger and larger. *XPPAUT* truncates this limit to `T=4000*dt`. For the present problem, since the kernel is

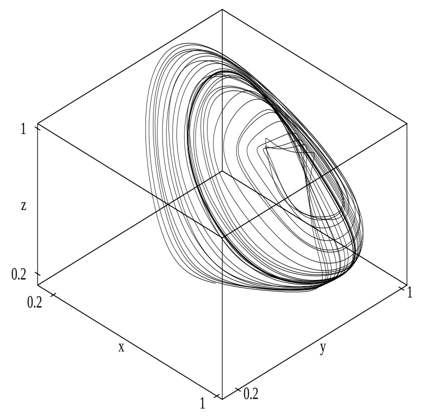


Figure 5.3. *Three-dimensional reconstruction of the chaotic integral equation.*

a Gaussian, this amount is a gross over-estimation. For the computer exp(−400) is zero. Thus, a dramatic increase in speed can be attained by setting the maximum number of points to 1000. To do this, click on `nUmerics Method`. Choose `Volterra` (which is really the only choice) and then accept all the defaults except `maxPoints`. Change this to 1000 (or even 400). Then click on **Esc** to get back to the main menu. Integrate this equation. Now let's plot it in three-dimensions. Click on `View axes 3D`. Fill the first three entries in, putting `X` on the `X-axis` and so on for `Y,Z`. Click on `Ok` and you will see an empty window. Click on `Window/zoom Fit` (**W F**) and the window is optimally chosen to include the entire graph. You should see something like Fig. 5.3

For fun, you might want to look at the bifurcations of this equation as the parameter r varies. There is a constant, $c_1(\tau)$ such that if $r < c_1(\tau)$, then $u = 0$ is a stable fixed point. At $r = c_1(\tau)$ there is a transcritical bifurcation to a nonzero positive fixed point. This remains stable until $r = c_2(\tau)$ where there is a Hopf bifurcation. Small amplitude oscillations emerge and then go through the usual period doubling cascade to chaos. You may want to prove some of these statements – this problem has never been analyzed.

5.1.3 Making 3D movies

While you have this file running and a nice three-d image, you can try to make a little movie. Click on `3d-params`. Ignore the first column of the dialog box, but fill in the second column as:

Movie(Y/N): Y
Vary (theta/phi): theta
Start angle: 45
Increment: 15
Number increments: 23

This tells *XPPAUT* to draw 23 frames of the screen image, rotating around the `theta` axis (around the z-axis) at 15 degree increments starting at 45 degrees. Before clicking OK, make sure that no other windows obscure the picture. Now click on **Ok** and sequence of 23 images will be drawn. These images have been stored in memory and can be played back. Click on `Kinescope Autoplay` . Choose `5` for the number of cycles and accept the default of `50` for the millisecond between frames. Your picture will be nicely rotated 5 times. Click on `Kinescope Make AniGif` and *XPPAUT* will create an animated GIF file called `anim.gif`. You can play this back using your web browser or using other software such as `xanim`. Repeat this exercise, varying `phi`, the other angle.

5.1.4 Singular integral equations

XPPAUT can solve integral equations with removable singularities e.g

$$J(t) = \int_0^t \frac{k(t-s)}{(t-s)^\mu} R(s)\, ds \qquad 0 \le \mu < 1. \tag{5.1}$$

Since $\mu < 1$, we can solve this equation by integrating by parts. See Linz, 1985, for a discussion of numerical methods that are used to solve singular and nonsingular Volterra equations. In the appendix, we describe the methods used in *XPPAUT* in more detail. The format for the expression (5.1) is `int[mu]{k(t)#R}`. The singular part is enclosed in the brackets [ ]. Here is an example ODE file demonstrating the syntax. The equation is

$$u(t) = \exp(-\frac{t}{2})\sqrt{t} + \frac{4}{3}\exp(-t)\sqrt{t^3} - \int_0^t \frac{\exp(-(t-s))}{\sqrt{t-s}} u(s)^2\, ds.$$

As can be easily verified, $u_T(t) = \exp(-t/2)\sqrt{t}$ is the true solution. Here is the ODE file

```
# a singular integral equation
u(t)=exp(-t/2)*sqrt(t)+(4/3)*exp(-t)*sqrt(t^3)-int[.5]{exp(-t)#u^2}
# and the true solution
aux utrue=exp(-t/2)*sqrt(t)
done
```

Run this and add the true solution to the graph to see that it does quite well.

5.1.5 Some amusing tricks

Periodic cicadas In his excellent book *Mathematical Biology*, Murray describes a model for periodic emergence of cicadas. I won't go through the derivation of the model but will briefly describe it. Certain types of insects seemingly emerge synchronously as adults after spending many years underground in a larval stage. Synchronous emergence means that all the insects come out in the same year. 13 and 17 year cicadas come out synchronously while 7 year cicadas emerge every year. The idea is that the grubs remain underground eating the roots of trees. When they emerge, predators are there to eat them. The underground roots have a limited carrying capacity. Here is the model. Let n_t be the number of adults emerging each year. At each year, a fraction μ survive. The amount of food that is remains to feed emerging adults is

$$c = d - \sum_{j=1}^{k-1} \mu^j n_{t-j+1}$$

where d is the maximum amount of food. The number of predators that are around at year t is

$$p_t = \nu p_{t-1} + a\mu^k n_{t-k}$$

The number of new grubs is the fecundity times the number that are left after predation:

$$n_t = \min(f \max(\mu^k n_{t-k+1} - p_t, 0), c).$$

What makes this problem "tricky" is that k is the parameter of interest. So this is a k^{th} order difference equation. Furthermore, you don't want to write all 17 equations to create a 17th order equation in normal form. We can avoid this by using the `delay` operator. Furthermore, we can even let the order, k be a parameter. Recall that `delay(x,k)` is $x(t-k)$ so that if t and k are integers, then `delay(x,k)` is just x_{t-k}. With these considerations, the ODE file for this problem is:

```
# cicada model
#
# here are the predators decay plus feeding on emerged adults
pt1=nu*p+a*mu^k*delay(n,k)
# amount of food left after taking into acct of all the grubs
c=max(d-sum(1,k-1)of((mu^i')*delay(n,i'-1)),0)
#
#
p(t+1)=pt1
# new number is what's left after predation and what could have eaten
n(t+1)=min(f*max(mu^k*delay(n,k-1)-pt1,0),c)
# set initial delay to 100
n(0)=100-0*t
aux ct=c
par a=.042,f=10,d=10000,nu=.95,mu=.95,k=7
# note you must set delay initial data to 100
@ meth=disc,dt=1,total=200,bound=1000000,delay=20
@ xhi=200,yhi=10000,yp=n,ylo=0
done
```

There are a few points to focus on here:

- I have defined the temporary variable `pt1` since the predation depends on this year and not the previous year. This way, I don't have to compute this twice.
- I have used the summation operator along with the delay operator to compute the sum $\sum_j \mu^j n_{t-j+1}$.
- I have set `dt=1` so that when data points are stored for the `delay` operator, they are properly referenced. I have also set `delay=20` so up to 20 year cicadas can be modeled.
- The initial condition statement `n(0)=100-0*t` sets the number of underground grubs from earlier generations to be 100. The apparently useless `0*t` is there to tell *XPPAUT* that this is a formula so that this must be a delay initial condition.

Run this and then try different values of k like 9,11,13, and 17. Which ones lead to synchronous emergence.

Fabry-Perot cavity. Here is a rather strange functional equation that combines a map with a continuous dynamical system. This problem arises in the analysis of something in laser physics called a pendular Fabry-Perot cavity. Juan M. Aguirregabiria provides this model as an example of his excellent dynamical systems program (**Dynamics Solver**) for Windows. Here are the equations:

$$\begin{aligned} x'' &= -x'/Q - x - x_0 + A\sin^2(\theta)|f|^2 \\ f(t) &= 1 + \cos\theta \exp(ix(t-\tau)) f(t-\tau) \\ \tau &= (r + (x(t) + x(t-\tau) - 2x_s)/c) \end{aligned}$$

where

$$x_0 = -x_s + A\sin^2(\theta)/(1 - \cos^2(\theta) - 2\cos(\theta)\cos(x_s)).$$

f is complex and the delay τ is implicitly defined. To avoid this difficulty, Aguirregabiria differentiates the τ equation to obtain:

$$\tau' = v(t)/(c + v(t-\tau))$$

where $v(t) = x'(t)$. We need to write $f = f_r + if_i$ in terms of its real and imaginary parts. Equations such as

$$f(t) = G(f(t-\tau))$$

present a challenge to *XPPAUT* since they are not ordinary differential equations. We will fool *XPPAUT* into treating them as Volterra equations without the integrals. However, *XPPAUT* will not use the Volterra solver if it cannot find any integrals, so we will define one that fools the program. With this in mind, here is the ODE :

```
# fp.ode
# fabry-perot cavity
init tau=1,x=1.08
# needed to fool XPP into using the Volterra solver
junk=int{0#x}
# ode for the delay
tau'=v/(c+delay(v,tau))
x0=-xs+a*sin(th)^2/(1-cos(th)^2-2*cos(th)*cos(xs))
x'=v
v'=-v/q-x-x0+a*(fr*fr+fi*fi)*sin(th)^2
volt fr=1+cos(th)*(cos(delay(x,tau))*delay(fr,tau)-sin(delay(x,tau))*delay(fi,tau))
volt fi=1+cos(th)*(sin(delay(x,tau))*delay(fr,tau)+cos(delay(x,tau))*delay(fi,tau))
par q=.11,xs=1.07,a=11.459,th=1,r=1.2221,c=120,eps=.1
@ dt=.05,total=200,trans=50,delay=4
@ vmaxpts=1
done
```

Most of this file is pretty straightforward. I fool *XPPAUT* into using the Volterra solver with the dummy integral `junk=int{0#x}`. The `volt` declaration tells *XPPAUT* to treat these two equations as though they were Volterra integral equations. I have added the statement `@ vmaxpts=1` which tells *XPPAUT* to change the maximum number of points stored by the Volterra integral solver to 1 which speeds everything up since no time is wasted evaluating the dummy integral. (You can also change this number within *XPPAUT* with the `nUmerics Method Volterra` command; type in 1 when prompted for MaxPoints.) Integrate the equations. Plot f_r, f_i, v in a nice three-d plot to see the chaotic behavior. Change the parameter a to 1 and solve it. Note the equations go to a fixed point. Increase a to see the periodic orbit. Note the period doubling between $a = 4$ and $a = 5$. The default value of a seems to be chaotic. (Note that in the above ODE file, we cannot write `fr(t) = ...` since the right-hand side contains nothing to tell *XPPAUT* that this is a functional equation rather than the definition of a simple function of time. *XPPAUT* looks for symbols like `int` to distinguish function definitions from functional equations.)

5.1.6 Exercises

1. Explore the Mackey-Glass equations:

$$x'(t) = -\gamma x(t) + \beta f(x(t-\tau))$$

   where $f(u) = u/(1+u^n)$. This is a model for circulating white blood cells. For humans, $\gamma = 0.1, \beta = 0.2, n = 10$ and the delay is $\tau = 6$. Integrate to 600, keeping output every 10. (`Total=600, Nout=10` in the `nUmerics` menu.) Be sure to start $x(0)$ away from 0 as $x = 0$ is a fixed point. Observe the small oscillations. Change τ to 20 and look at the new behavior. Try this: set $\tau = 1$ and integrate to the fixed point. Click on `Initialconds Last` to be sure you are on it. Determine the stability and notice that the fixed point

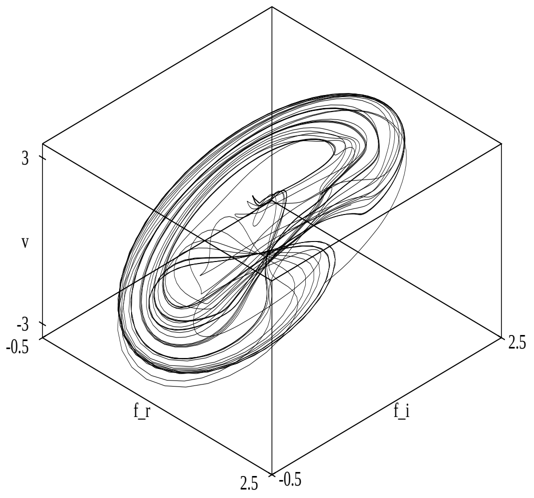


Figure 5.4. *Solution to the Fabry-Perot equations in the chaotic regime.*

is stable. Click on `sIng pts` `Range` and choose τ as the range parameter going from 1 to 6 in 10 steps. Click on `Ok` . You will see in the **Equilibrium Window** that the rest state has lost stability. Look at the console window (if you stated *XPPAUT* from a command line). You will see a list of complex numbers. Each is the eigenvalue with maximal real part for the particular parameter. Note the change in sign of the real part.

2. Solve the state-dependent delay equation:

$$x'(t) = x(t)(2 - x(t))(1 - x(t - s))$$

where $s = \max(0, px)$ where p is a positive parameter. Choose $x(0) = .1$ and vary p between 1 and 5. Make sure you choose a large value for the maximal delay (say, 10). For what values of p is the fixed point stable?

3. Solve the neural network model:

$$\frac{du}{dt} = r[-u(t) + (1 - \int_{t-1}^{t} u(s)\,ds) f(u(t))]$$

where $f(u) = 1/(1 + \exp(-8(u - .333)))$ and r is a parameter. Hint: Write

$$\int_{t-1}^{t} u(s)\,ds = \int_{0}^{t} \mathrm{heav}(1 - (t - s))u(s)\,ds \qquad \text{for} \quad t > 1,$$

so that it is a convolution of u with the Heaviside function. Try $r = 3$ and then $r = 6$. For what values is the rest state apparently stable?

4. Solve
$$u(t) = \sqrt{t} + \frac{\pi}{2}t - \int_0^t \frac{u(s)}{\sqrt{t-s}}\,ds$$
and compare the solution to the analytic solution, $u(t) = \sqrt{t}$.

5. Solve another integral equation due to Mike Mackey:
$$\frac{dv}{dt} = \Gamma(e-v) - \beta_0 g[\int_0^t \max(v(t') - \theta(t-t'), 0)\xi(t-t')\ dt']$$
where
$$\theta(t) = \frac{1}{(t+\epsilon)^3} \quad \xi(t) = \frac{\text{heav}(t-t)\text{heav}(t_{max}-t)}{t_{max}-1} \quad g(v) = \frac{v}{1+v^3}.$$
The parameters are $\epsilon = .001$, $\Gamma = 10, e = 1.6$, $\beta_0 = 600$, $f_0 = 8$, $t_{max} = 1.625$. Integrate between 0 and 20 with `dt=0.01`. Here is the ODE file

```
#mackey2.ode
#
xi(t)=heav(t-1)*heav(tmax-t)/(tmax-1)
#
th(t)=1/(t+eps)^3
par eps=.001
#
g(y)=y/(1+y^n)
#
v'=gamma*(e-v)-beta0*g(f0*int{max(v-th(t-t'),0)*xi(t-t')})
p gamma=10,n=3,f0=8,beta0=600,tmax=1.625,e=1.6
init v=.9
@ TOTAL=20,DT=0.01,XLO=0,XHI=20,YLO=-30,YHI=10,VMAXPTS=400
done
```

5.2 Stochastic equations.

XPPAUT allows you to simulate a variety of random processes including continuous Markov processes and Brownian motion. *XPPAUT* has two functions that return random values, `normal(a,b)` which returns a normally distributed random variable with mean a and standard deviation b. The function `ran(c)` produces a uniformly distributed random variable from the interval $(0, c)$. *XPPAUT* uses the general purpose random number generator described in *Numerical Recipes in C*, `ran1()`.You can always start the random number generator at the same value by picking the same seed with the `nUmerics stocHastic Seed` (**U H S**) command.

Suppose you want to simulate the system:
$$dx = -f(x)dt + d\xi.$$

Then here is the correct Euler discretization:

$$x(t + \Delta t) = x(t) - f(x)\Delta t + \sqrt(\Delta t)\phi$$

where ϕ is normally distributed with zero mean and unit variance. *XPPAUT* saves you the trouble of doing this by introducing "wiener" parameters. These are normally distributed numbers with mean zero and variance, $\sqrt{\mathrm{dt}}$ where dt is the numerical time-step. Thus, the ODE file would contain these statements:

```
# code fragment
wiener w
x'=-f(x)+sigma w
```

This is just shorthand for the scaled noise.

For example, consider the case of pure Brownian motion, $f(x) = 0$.

$$dx = \sigma d\xi$$

The ODE file below is sufficient:

```
# brown.ode
#  the wiener process
wiener w
x'=sig*w
par sig=1
@ meth=euler
@ ylo=-20,yhi=20
done
```

Note that whenever you do simulations with this type of noise, it is important that you use Euler's method as *XPPAUT* does not have any special integrators for noisy simulations. You can use other integrators like Runge-Kutta, but the noise is held constant for the entire interval from t to $t + dt$.

Try this ODE file. Here is an experiment to do. Look at the mean and variance of 1000 sample paths. To do this, click on `nUmerics stocHastic Compute` (**U H C**) and fill in the first 4 lines of the dialog box as follows:

`Range over:` X
`Steps:` 1000
`Start:` 0
`End:` 0

and click on **Ok** . Once the simulation is done (it will take a few seconds or more depending on the computer speed), click on `stocHast Mean` (**H M**) and exit back to the main menu. Click on `Erase` and `Restore` (**E R**) to erase the screen and redraw the mean. The mean will hover very close to 0 with only a small amount of deviation. Save this curve in memory using the `Graphics Freeze Freeze` command. Choose `1` for the color (red) and click **Ok** . As is well known for brownian motion, the variance grows linearly with time. To see this, click on `nUmerics stocHast Variance` and `Exit` back to the main menu. Click on `Restore` to draw the variance. It is almost a perfectly straight line with slope 1. (Click on `Graphics Add curve` and add the curve with `t` on both the X and Y axes. This represents a line with slope 1 and is quite close to your plot of the variance.)

5.2.1 Markov Processes

XPPAUT is able to simulate continuous Markov processes with transition rates that depend on other state variables in the differential equations. An obvious example is the simple two state voltage-dependent channel model. Suppose you have a patch of membrane with a single sodium channel that opens at a rate $\alpha(V)$ and closes at a rate $\beta(V)$. Let the open state have conductance 0. Then, the equation has the form:

$$C\frac{dV}{dt} = -g_L(V - V_L) + I - gz(V - V_{Na})$$

where z is a random variable that is 1 for the conducting state and 0 for the nonconducting state. z flips from 0 to 1 in the interval t to $t + dt$ with probability $\alpha(V)dt$ and flips from 1 to 0 with probability $\beta(V)dt$. *XPPAUT* simulates this by computing the probability of flipping at each time step and then changing the variable z. z is always regarded as a nonnegative integer. In order to define a Markov process, you must define the transition matrix. For the present model, the matrix is defined in *XPPAUT* with the statement

```
markov z 2
{0} {al(v)}
{beta(v)} {0}
```

where `al(v),beta(v)` are the rate functions. In *XPPAUT* each entry in the matrix is delimited by a pair of curly brackets { }. The diagonal entries are always 0 since they are never actually computed — they are just the probability of not changing and equal 1 minus the rates of the rest of the row. The complete *XPPAUT* file for simulating the single channel is:

```
# channel.ode - random channel model using HH alpha and beta
#
v' =(I-gl*(v-vl)-gna*z*(v-vna))/c
par I=0,gl=.1,gna=.001,vna=55,vl=-65,c=1,phi=1
init v=-65,z=0
markov z 2
{0} {al(v)}
{beta(v)} {0}
al(v)=phi*.1*(v+40)/(1-exp(-(v+40)/10))
beta(v)=phi*4*exp(-(v+65)/18)
aux ina=-gna*z*(v-vna)
@ total=100,meth=euler
@ xlo=0,xhi=100,ylo=-66,yhi=-64
done
```

The file is pretty self-explanatory. I have added the current `ina` as something extra to plot. The method of integration is Euler's method. If you want to simulate thousands of channels, it is much better to use a method due to Gillespie. However, for small numbers of channels, one can define a bunch of Markov variables.

Another example of a continuous Markov model is due to Tom Kepler. Here we take the standard logistic equation for growth of a single species. We assume that there is a random mutation whose probability depends on the population of the first species. This mutation competes with the first and eventually dominates. The equations that Kepler proposes have the form:

$$x_1' = x_1(1 - x_1)$$

before the new mutant appears and

$$x_1' = x_1(1 - x_1 - x_2) \quad x_2' = ax_2(1 - x_1 - x_2) + \epsilon x_1$$

after it appears. *XPPAUT* does not allow you to change the dimension of the dynamical system, so we trick the program by introducing both equations, but allowing no changes until the mutation occurs. We assume that the rate of the mutation is ϵx_1, that is, it is proportional to the population of species 1. Here is the *XPPAUT* file:

```
# Kepler model kepler.ode
#
init x1=1.e-4,x2=0,z=0
#
par eps=.1,a=1
x1' = x1*(1-x1-z*x2)
x2'= z*(a*x2*(1-x1-x2)+eps*x1)
# the markov variable and its transition matrix
markov z 2
{0} {eps*x1}
{0} {0}
@ total=100,xhi=100,ylo=0,yhi=1.2
@ nplot=2,yp2=x2
done
```

Here are some notes on the ODE file: (i) I multiply the right-hand side of `x2'` and also `x2` in the `x1'` equation by `z` which is zero until the mutation comes alive; (ii) the mutation never turns off so the second row of the transition matrix is all 0's; (iii) I have added the line `nplot=2,yp2=2` so that *XPPAUT* also plots the second population against time.

Run this. Here is a question: What is the expected value of z as a function of t? We can answer this through a Monte Carlo simulation. As with the Brownian motion problem, we can perform a repeated simulation and take the mean and variance. Use the `nUmerics stocHast Compute` option to run this simulation 100 times (choose `z` as the quantity and set the start and end to 0). (It will take several seconds for this computation.) While still in the `nUmerics` menu, type

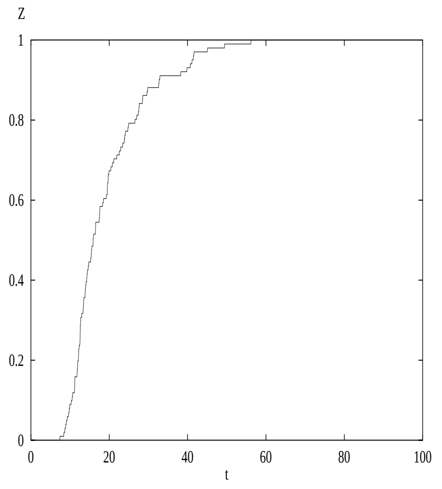


Figure 5.5. *The probability of existence of the mutant species as a function of time.*

`stocHastic Mean` to load the mean values of all the variables. Escape to the main menu and plot the mean of `z` as a function of time. You should see something like figure 5.5

5.2.2 Gillespie's method

The above method of simulating a Markov process is good for small numbers of such events. However, if you want to simulate thousands of molecules, using the `markov` declaration is out of the question. Furthermore, this method of simulating the Markov process is only an approximation to the real continuous time process. Events can only occur at values of the time step `dt.` Dan Gillespie (1977) devised a method which is exact and can be used to simulate thousands of random processes. The method is basically to use the ideas of mass action and thus simulate each reaction at the single molecule level. This method has been used to study channels in active membrane models as well as a number of chemical processes. We illustrate it here on a simple chemical reaction and then with the Brusselator, a classic chemical oscillator. Consider the reaction:

$$X \longrightarrow Z$$

for the degradation of the molecule X at rate r. Suppose that there are N molecules of X. Since Markov processes are exponentially distributed, the time of the next reaction taking place is a random variable with probability $\exp(-at)$ where $a = rN$ is the rate. Thus, to compute when the next reaction takes place, we merely draw a uniform random number, s, between (0,1) and take

$$t_{next} = -\frac{1}{a}\ln(s) = \frac{1}{a}\ln(1/s).$$

The reaction reduces the number of molecules of X by 1. If there are several reactions, then the total rate a_0 is the sum of all their rates. A random number, s_1 is drawn to determine the next time

$$t_{next} = -\frac{1}{a_0}\ln(s_1).$$

Then, another random number is chosen to determine which reaction takes place. Let s_2 be the second random number uniformly drawn from the interval $(0, a_0)$. Let

$$P_j = \sum_{k=1}^{j} a_k$$

where a_j are the m reaction rates. Reaction 1 occurs if $s_2 < P_1$, reaction 2 occurs if $P_1 \le s_2 < P_2$, and so on. This method is exact; there is no inherent time step, the reactions occur when they naturally would. Thus, the iteration for this process counts reaction steps rather than time.

We first show the ODE file for the simple degradation reaction since there is no need to determine the reaction; there is only one.

```
# gillespie.ode
# this implements gillespies algorithm
#  X -> Z  at rate c
par c=.5,xin=1000
init X=1000,tr=0
a0=c*X
tr'=tr+log(1/ran(1))/a0
X'=max(X-1,1)
aux cts=xin*exp(-c*tr)
@ bound=100000000,meth=discrete,total=1000
@ xlo=0,xhi=10,ylo=0,yhi=1000,xp=tr,yp=x
done
```

We use the discrete integrator since the reactions are discrete events. We let `X` be the number of molecules of X. The total rate is $a_0 = cX$ and the time until the next reaction is computed by drawing a random number. X is decremented by 1. We don't let X fall below 1 since then $a_0 = 0$ and the reaction stops. We initialize for 1000 reaction steps. We plot X against the variable t_r which represents real time. I have added an auxiliary variable which represents the solution to the mean field equation

$$\frac{dX}{dt} = -cX \quad X(0) = 1000.$$

Run this and graph the continuous approximation along with it. Note that they are quite close. Try this again but let $X = 10000$ initially and set `xin=10000`. Only plot every tenth reaction point (Click on `nUmerics nOut` and change this to 10. Also change `Total` to 10000 and then click on `Esc-exit` and integrate again.) They are even closer.

Consider the classical Brusselator chemical reaction

$$\begin{aligned} X_1 &\longrightarrow Y_1 \\ X_2 + Y_1 &\longrightarrow Y_2 \\ 2Y_1 + Y_2 &\longrightarrow 2Y_1 \\ Y_1 &\longrightarrow Z \end{aligned}$$

with rates c_1, c_2, c_3, c_4 respectively. Only Y_1, Y_2 vary with X_1, X_2 fixed. The rates used to compute the probabilities are

$$\begin{aligned} a_1 &= c_1 X_1 \\ a_2 &= c_2 X_2 Y_1 \\ a_3 &= c_3 Y_2 Y_1 (Y_1 - 1)/2 \\ a_4 &= c_4 Y_1. \end{aligned}$$

(Note the third reaction; once one Y_1 is chosen there are $Y_1 - 1$ remaining and we must divide by 2 since we are counting pairs of "Y's" and don't want to count the same pair twice.) Here is the corresponding *XPPAUT* file

```
# gillesp_bruss.ode
# gillespie algorithm for brusselator
#
# x1  -> y1 (c1)
# x2+y1 -> y2+Z (c2)
# 2 y1 + y2 -> 3 y1 (c3)
# y1 -> Z2 (c4)
par c1x1=5000,c2x2=50,c3=.00005,c4=5
init y1=1000,y2=2000
#  compute the cumulative reactions
p1=c1x1
p2=p1+c2x2*y1
p3=p2+c3*y1*y2*(y1-1)/2
p4=p3+c4*y1
# choose random #
s2=ran(1)*p4
z1=(s2<p1)
z2=(s2<p2)&(s2>=p1)
z3=(s2<p3)&(s2>=p2)
z4=s2>p3
# time for next reaction
tr'=tr-log(ran(1))/p4
y1'=max(1,y1+z1-z2+z3-z4)
y2'=max(1,y2+z2-z3)
@ bound=100000000,meth=discrete,total=1000000,njmp=1000
@ xp=y1,yp=y2
@ xlo=0,ylo=0,xhi=10000,yhi=10000
done
```

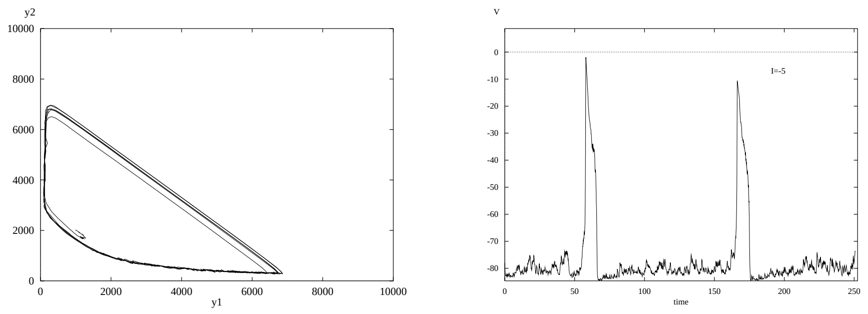


Figure 5.6. *The Gillespie method applied to the Brusselator(left) and to a membrane model.*

Since X_1, X_2 are held constant, I have combined the products c_1X_1 and c_2X_2 into single parameters. Next, I compute the cumulative sum P_j. Note that $P_4 = a_0$ since it is the sum of all the rates. I choose a random number s_2 between 0 and a_0. Now comes the tricky part. I define 4 numbers z_j. These are 1 if reaction j takes place and 0 otherwise. Thus if $s_2 < P_1$ then reaction 1 takes place. If $P_1 \le s_2 < P_2$ then reaction 2 takes place and so on. Next, we determine the time of the next reaction. Then Y_1, Y_2 are incremented or decremented according to the reaction. For example reactions 1 and 3 produce an increase by one of Y_1 and reactions 2 and 4 lead to a loss of Y_1. If you run this in *XPPAUT* , and plot the (Y_1, Y_2) phase-plane you will get a figure something like the left side of Figure 5.6.

As a final example, we consider a simple model neuron which has three channels: a leak channel, a potassium channel, and a persistent sodium channel. The leak channel has a fixed conductance, the potassium channel has the usual voltage gated conductance as does the sodium channel. However, here, we will assume that the number of sodium channels is finite so that the random opening and closing of them becomes important. Here is the system of equations:

$$C\frac{dV}{dt} = I - g_L(V - E_L) - g_K w(V - E_K) - g_{Na}x(V - E_{Na}) \tag{5.2}$$

$$\frac{dw}{dt} = \frac{w_\infty(V) - w}{\tau_w(V)} \tag{5.3}$$

and x is the fraction of open sodium channels. Let o be the number of open channels and $c = N - o$ be the number of closed channels where N is the total number of channels. The transition between open and closed states is governed by voltage-dependent rates:

$$c \longrightarrow o$$
$$o \longrightarrow c$$

with rates $a(V)$ and $b(V)$ respectively. We can implement the Gillespie method on the transitions between open and closed just as we have in the previous examples.

However, we now have, in addition, a pair of ordinary differential equations to deal with. Thus, as each reaction advances the clock by an amount Δt, we must integrate the equations by this amount of time. *XPPAUT* does not "know" about the Gillespie method, so we must approximate the integration by a discrete time iteration. The obvious choice would be to use Euler's method to advance the voltage and the potassium gating variable, w. However, Euler's method is fraught with instabilities if the time step is too big. Thus, we introduce another first order method that is often used to solve membrane models: *exponential Euler.* Consider the differential equation:

$$y' = a - by$$

where a and b may be complicated functions of other variables and time. For the period of time Δt we assume that a, b are approximately constant. Then we can exactly integrate this equation for an amount of time Δt:

$$y(\Delta t) = a/b + (y(0) - a/b)\exp(-b\Delta t).$$

This trick is called the exponential Euler method. The advantage of it is that if b is very large, the algorithm remains stable without Δt being very small. (See also Duman and Mascagni in Koch & Segev, 1989.) It is just as inaccurate as Euler's method but is not unstable. Since both the voltage equation and the potassium gating variable can be written as $y' = a - by$ where $b > 0$, we can use this method to advance both variables. Thus, the method for integrating the equations is to first figure out the time of the next reaction giving Δt and update the channels using the Gillespie method and update the deterministic equations using the exponential Euler method:

$$y_{n+1} = a/b + (y_n - a/b)\exp(-b\Delta t).$$

Thus, the *XPPAUT* implementation of the stochastic membrane model is

```
# persistent sodium current ala Gillespie
# napgill.ode
# we use the exponential integration method
#   c -> o  rate a
#   o ->c   rate b
# definition of a,b
# a=xinf*tau^-1
# b=tau^-1 -a
xinf=.5*(1+tanh((v-vxt)/vxs))
tauxi=phix*cosh((v-vxt)/(2*vxs))
a=xinf*tauxi
b=tauxi-a
# the cumulative probabilities
p1=a*(xn-o)
p2=p1+b*o
# which reaction?
s2=ran(1)*p2
z1=(s2<p1)
```

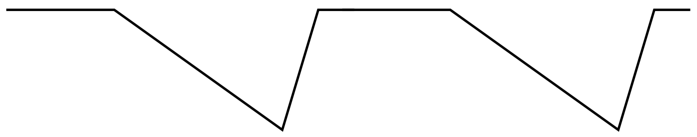


Figure 5.7. *An asymmetric periodic potential*

```
# when is the next reaction?
dt=-log(ran(1))/p2
# the fraction of open channels
x=o/xn
# the total conductance
gv=gl+gk*w+gna*x
# the reversal potential
vrev= (gl*vl+gk*w*vk+gna*x*vna+i)/gv
# that is,  v' = -gv (v-vrev)
init v=-78,w=.3
# here is the exponential method
v'=vrev+(v-vrev)*exp(-gv*dt)
w'=winf(v)+(w-winf(v))*exp(-dt/tauw(v))
# update reaction time
tr'=tr+dt
# and the channels
o' = max(1,o+2*z1-1)
# parameters
par xn=100
par i=0,gk=10,gl=.1,gna=2.5,vk=-85,vl=-70,vna=120
par vxt=-52,vxs=15,vwt=-65,vws=20,phix=10,phiw=.1
# functions
winf(v)=.5*(1+tanh((v-vwt)/vws))
tauw(v)=1/(phiw*cosh((v-vwt)/(2*vws)))
# some XPP settings
@ meth=discrete,total=20000,bound=10000000,njmp=10
@ xlo=0,xhi=200,xp=tr,yp=v,yhi=20,ylo=-85
done
```

Run this. Change the total number of channels, `xn`, as well as the total number of reaction steps. (Since more channels means that the rates of reaction happen at shorter time intervals, if you double the number of channels, you want to double the number of reaction steps. In the numerics menu, change `Total` and also change `Nout` doubling that as well.) The right panel of figure 5.6 shows the result of one such calculation.

5.2.3 Ratchets and games

Brownian ratchets of various sorts have been the subject of much recent interest. Many molecular motors work on this principle. The idea is that an asymmetric potential is periodically varied and combined with Brownian motion. The net effect of these two random processes is the performance of work. Let $V(x)$ be a potential function. Assume that it is periodic on the line with period 1 and that it is asymmetric (see the figure 5.7). Let z be a two state Markov process that determines the magnitude of the potential — when $z = 0$ the potential has zero magnitude and when $z = 1$ it is maximal. Finally assume a certain amount of Brownian motion. Then when the potential is on, the probability distribution builds up proportionally to $P(x) = \exp(-V(x)/\sigma)$. When the potential is turned off, there is diffusional drift but it favors one direction more than the other because of the asymmetric potential. The result is that there is a net flux in one direction. Many molecules work in this fashion by changing configuration (due to the hydrolysis of ATP) to produce in effect a molecular ratchet driven by the noise. There are many realistic models of this but they all work on the same principle illustrated by the above example. The equations for this are:

$$dx = -zV'(x)dt + \sigma d\xi,$$

where z is a two state Markov process that has switching rates α, β for $0 \to 1$ and $1 \to 0$ respectively. Here is the *XPPAUT* file, `ratchet.ode` :

```
# a simple thermal ratchet
wiener w
par a=.8,sig=.05,alpha=.1,beta=.1
# piecewise linear potential with slope
# 1 from 0 to a and slope -a/(1-a) from a to 1
#  f = -V'
f(x)=if(x<a)then(-1)else(a/(1-a))
x'=z*f(mod(x,1))+sig*w
# z is two states
markov z 2
{0} {alpha}
{beta} {0}
@ meth=euler,dt=.1,total=2000,njmp=10
@ xhi=2000,yhi=8,ylo=-8
done
```

The program is pretty self-explanatory. Since I only define the function $f(x)$ on the interval $[0, 1]$ I must consider x modulo 1 to make it periodic. Run this and you will see the discrete steps that x makes as it ratchets downward. Note that this particular ratchet tends to move toward negative x. Run this 100 times and compute the average trajectory. (Use the `nUmerics stocHastic Compute` command and then the `Mean` item – as above.) Change `a` from 0.8 to 0.2. Run it again. Change `a` to 0.5 and run it once more. Note that by varying a you can make the ratchet go in any direction at a variety of velocities.

A surprising game

In Harmer and Abbott (1999), an intriguing paradox was displayed and subsequently analyzed. In this paper, two games were played. Both games are losers in that repeatedly playing them leads to a net loss. However, when the games are played sequentially or chosen randomly with 50% probablility, then they produce a winning strategy. Game one consists of flipping a coin that has a slightly less than 50% chance of winning. Thus over the long haul, you lose. Game two works like this. If the amount of money you have is a multiple of three than you flip a very badly biased coin. If the amount of money you have is not a multiple of three, then you flip a coin with a good probability of winning. Ultimately, the very badly biased coin costs you and this game is a long term loser. Now consider the third game. At each turn you flip a fair coin. If heads, then play game 1 and if tails play game 2. This turns out to be analogous to the ratchet we described above but it is a discrete analogue. Let's write an ODE file for each of these games. Here is game 1:

```
# game1.ode: this plays game 1
game(p,e)=if(ran(1) < (p-e))then(1)else(-1)
par eps=.005,pa=.5
ca'=ca+game(pa,eps)
@ total=100,meth=discrete,dt=1
done
```

The function `game1(p,e)` returns ± 1 according to whether or not you win. The variable `ca` tracks the amount of cash you have. The method is discrete. You play 100 games in each simulation. Since the probability of winning is 0.495, over the long haul, this is a losing game. Simulate this 1000 times and look at the mean value of the amount of cash as a function of the number of games played. (Use `ca` as the variable to vary and set the maximum and minimum to 0).

Game 2 is more subtle. Here is the ODE file

```
# game2.ode: this plays game 2
game(p,e)=if(ran(1) < (p-e))then(1)else(-1)
par eps=.005,p2=.1,p3=.75,m=3
p=if(mod(ca,m)==0)then(p2)else(p3)
ca'=ca+game(p,eps)
@ total=100,meth=discrete,dt=1
done
```

As above, we play 100 games and look at the amount of cash we have. Simulate this 1000 times and you will see that it is clearly a losing game. You lose about 1.5 per 100 games.

Now finally, we write a simulation for alternating between the two games. Let p_a be the probability from game 1 and let p_1, p_2 be the two probabilities from game 2. Let γ be the probability of choosing game 1 at any point in time. Let ϵ be a small parameter that makes the games unfair. It is elementary to show that game 1 is a loser if $1 - p_a + \epsilon > p_a - \epsilon$. A less routine calculation shows that game 2 is a loser if

$$(1-p_2+\epsilon)(1-p_3+\epsilon)^2 > (p_2-\epsilon)(p_3-\epsilon)^2.$$

You can check that for the parameters chosen, both games are losers. However game 3 is a winner and thus illustrates what is called Parrondo's paradox if the following conditions are met:

$$\begin{aligned}(1-q_2)(1-q_3)^2 &< q_2q_3^2\\ q_2 &= \gamma(p_a-\epsilon)+(1-\gamma)(p_2-\epsilon)\\ q_3 &= \gamma(p_a-\epsilon)+(1-\gamma)(p_3-\epsilon).\end{aligned}$$

```
# game3.ode
# this plays the combined game with probability 1/2 each
# Parrondo's paradox
game(p,e)=if(ran(1) < (p-e))then(1)else(-1)
par eps=.005,p2=.1,p3=.75,m=3,pa=.5,gamma=.5
# probability for game 2 if it is played
pg2=if(mod(ca,m)==0)then(p2)else(p3)
# probability for game 1 is it is played
pg1=pa
# probability for this game
p=if(ran(1)>gamma)then(pg2)else(pg1)
# now play the game and increment or decrement our cash
ca'=ca+game(p,eps)
@ total=100,meth=discrete,dt=1
done
```

As in the above two games, run this simulation 1000 times. You will find that you net on average about 1.2 after 100 games. Figure 5.8 illustrates the effects of playing all three games 10000 times. Here is the combined simulation:

```
# parrondo.ode
# this plays the combined game with probability 1/2 each
# Parrondos paradox
game(p,e)=if(ran(1) < (p-e))then(1)else(-1)
par eps=.005,p2=.1,p3=.75,m=3,pa=.5,gamma=.5
# game 1 alone
ca_1'=ca_1+game(pa,eps)
# game 2 alone
pg2p=if(mod(ca_2,m)==0)then(p2)else(p3)
ca_2'=ca_2+game(pg2p,eps)
# game 3
pg2=if(mod(ca_c,m)==0)then(p2)else(p3)
pg1=pa
p=if(ran(1)>gamma)then(pg2)else(pg1)
ca_c'=ca_c+game(p,eps)
@ total=100,meth=discrete,dt=1
done
```

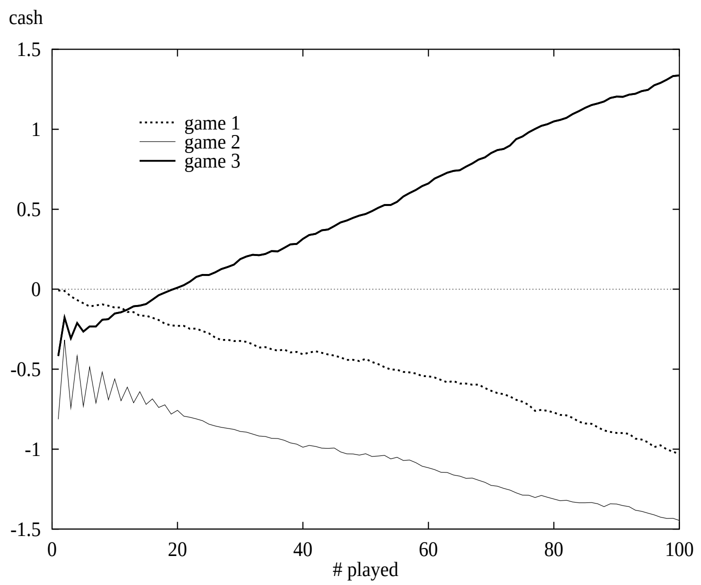


Figure 5.8. *Illustration of Parrondo's paradox – games 1 and 2 are losers but played together become winners.*

5.2.4 Spike-time statistics

In many neurobiological applications, we are interested in the spike-time statistics of single neurons and populations of neurons. *XPPAUT* has a number of features that let one simulate and analyze equations with noisy inputs. We will look at histograms for Poisson processes with and without refractoriness, post-stimulus histograms, and spike-time autocorrelation functions.

Post-stimulus time histograms

When an experimentalist records from a neuron that is receiving some input, the spike time histogram (PSTH) is often used as a measure of the response. One looks at the distribution of spike times as a function the onset of the input. Many trials are run and a histogram of the times is made. I present a simple example here. A simplified model neuron (the "theta" model) which like the integrate and fire, lies on a circle. Here are the equations we will simulate:

$$u' = 1 - \cos u + (1 + \cos u)(I_0 + \sigma w(t) + f(t))$$

where $w(t)$ is normally distributed noise, I_0 is the bias, and $f(t)$ is the stimulus. We will use $I_0 = -0.2, \sigma = 0.25$ and

$$f(t) = 0.4t\exp(-t/8).$$

By definition, the "theta" model spikes when u crosses π. It is defined modulo 2π as well so when it hits 2π it is reset to 0. Here is the ODE file, `psth.ode`

```
# psth.ode
# using an alpha function input and the theta model with noise
wiener w
par i0=-.2,sig=.25,amp=.4,tau=8
init u=-.84
f(t)=amp*t*exp(-t/tau)
@ meth=euler,total=50,dt=.1
u'=1-cos(u)+(1+cos(u))*(i0+sig*w+f(t))
global 1 u-2*pi {u=0}
#
done
```

I have initialized u to its resting value assuming that there are no inputs or noise. We want to record the times t_j at which a spike occurs; that is when $u(t)$ crosses π so we will create a Poincare section. Click on `nUmerics Poincare map Section` and choose `u` as the variable and `3.14159` as the section. `Esc` to the main menu and integrate. Look in the **Data Viewer** and you will see a few times recorded. We will now integrate this 100 times and store all of the spike times. Click on `Initialconds Range` and choose `sig` as the parameter, with `.25` and both the starting and ending values. Choose `100` steps and **most importantly** choose `N` for the `Reset Storage` option so that the browser is not reset after every run. Click `OK` and it will start. It may take a few minutes to finish. Once you are done, you will have a list of about 700 times, an average of seven spikes per stimulus. Now will will make a histogram. Click on `nUmerics sTochastic Histogram` . Choose 100 for `Number of bins`, 0 for `Low`, 50 for `High`, `T` for the variable to histogram, and `Enter` for the condition. In a second the histogram is computed. `Esc` to the main menu and plot `U vs t` since the first column contains the bins and the second the number per bin. You should see a histogram with several peaks trailing off to zero. (See Fig 5.9, left). Repeat this with more or less noise to see how the histogram changes. This technique can be applied to any model neuronal system in which the spike is clearly defined by some crossing event.

We can quantify this apparent periodicity by looking at the spike-time autocorrelation function. This makes a histogram of the differences between spike times. Click on `nUmerics stocHast Data` to get the list of spike times back into the **Data Viewer** . Click on `stocHast Stat autOcor` to make an autocorrelation. Choose 200 bins ranging from -50 to 50 with `t` as the variable and replot the first two columns. You will see a clear periodicity. You can see it clearer if you narrow the range of the histogram from -20 to 20. This is the **spike-time autocorrelation function.** There is a clear periodicity with a peak distance of about 3.

A Poisson process with refractoriness

Recall that a Poisson process can be modeled using Markov variables . here we will simulate a neuron which spikes randomly with an exponential distribution of the interspike intervals but which has an exponentially decaying refractory period. Thus the rate r is $r_{max}(1-s)$ where $s=1$ when there is a spike and s decays exponentially with a time constant τ. Here is the ODE file, `poisref.ode`

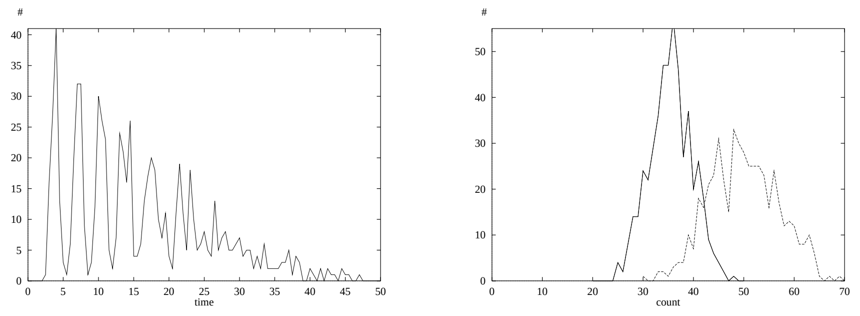


Figure 5.9. *PSTH for the noisy theta neuron (left) and the spike count distributions for a Poisson process with rate* 0.05 *and the same process with a refractory period.*

```
# poisref.ode
# poisson process with refractory
# period
# s0=0 for no refractory
r=rmax*(1-s0*s)
# define a two-step Markov process with rate r
markov z 2
{0} {r}
{1} {0}
# define a flag so that when z jumps past 1/2
# reset it to 0, turn on refractory period and
# increment the count
global 1 z-.5 {s=1;z=0;count=count+1}
# exponential decay of the refractory period
s'=-s/tau
count'=0
par rmax=.05,tau=10,s0=0
# set it up so only the value at the end of an expt is computed
@ meth=euler,dt=.1,total=1000,bound=1000000,nout=500,trans=1000
done
```

I have added a number of comments that explain what is going on. We will simulate this over many trials and look at the spike count distribution. Time is in milliseconds so that with a rate 0.05 we expect about 50 spikes per trial. We will simulate 500 trials. Click on `Initconds Range` and range over `s` from 0 to 0 in 500 steps and set `Reset Storage` to `N`. This will take a few minutes. When done, you will have the standard Poisson process with no refractory period. The **Fano** factor is the variance of the count divided by the mean. Click on `nUmerics stocHast Stat Mean/Dec` and choose `count`. You will get the mean spike count of about 50 and a standard deviation of around 7. The Fano factor is about $7^2/50$ which is close to the theoretical value of 1 for a Poisson process. Make a histogram of the counts. Click on `stocHast Histogram` with 40 bins of the variable `count` with a range between 30 and 70. Plot this and it looks close to a Gaussian. (A rather jagged one). Freeze this if you want to compare it to the next case. Now change `s0=1` to turn on the refractory period. Repeat the range integration. (You needn't change any of the dialog entries.) Find the Fano factor and make a histogram. I found a mean of 35.5 and a deviation of 4.25 with a Fano factor of 0.5 which is much more regular than Poisson. Make a histogram of the counts with 30 bins in the range 20 to 50 on the variable `count` and you will see a much narrower histogram (cf Fig 5.9). The refractory period makes the spike count more regular.

5.2.5 Exercises

1. Implement the Gillespie algorithm for the Lotka-Volterra equations:

$$\begin{aligned} X_1 + Y_1 &\longrightarrow 2Y_1 \\ Y_1 + Y_2 &\longrightarrow 2Y_2 \\ Y_2 &\longrightarrow Z \end{aligned}$$

   where $c_1X_1 = 10, c_2 = 0.01, c_3 = 10$ and initially $Y_1 = 1000, Y_2 = 1000$. Note that only Y_1, Y_2 are variables and X_1 is held fixed.

2. Implement the fully deterministic version of the neural model by adding the following differential equation for x the fraction of sodium channels:

$$\frac{dx}{dt} = (x_\infty(V) - x)\tau_x^{-1}(V)$$

   where x_∞ and τ_x^{-1} are defined in the ODE file as `xinf` and `tauxi` respectively. Figure out the minimum current for which oscillations occur. For low channel numbers, the stochastic model will fire spikes occasionally even though the current is quite far below the critical value for repetitive oscillations. Finally, set $I = 10$ in the deterministic model and compute the period of the spikes. Do the same for the stochastic model using 100, 1000, and 4000 channels.

3. A recent PRL paper showed that you could get directed motion with multiplicative noise if it is correlated with additive noise. Here is an example system:

$$\begin{aligned} x' = &-\sin(x/2)(J_1 + \sqrt(D_1)\lambda_1 y_1) - \sin(x + x_0)(J_2 + \sqrt(D_2)y_2\lambda_2) \\ &+ \sqrt(D)(\lambda_1 y_1 + \lambda_2 y_2 + \sqrt(1 - \lambda_1^2 - \lambda_2^2)w) \end{aligned}$$

   where y_1, y_2, w are normally distributed (properly scaled) random variables. Note that the additive noise is correlated with the multiplicative noise if $\lambda_j \neq 0$. Write an ODE file and simulate it up to $t = 500$ 200 times and plot the mean value of x as a function of time for $D = 3, J_1 = 0.7, j_2 = 1, D_1 = D_2 = 0.3, x_0 = \pi/2$ and $\lambda_1 = \lambda_2 = 0.3$. Repeat this with $\lambda_1 = \lambda_2 = 0$. Note the lack of any drift in the latter case.

5.3 Differential algebraic equations.

XPPAUT has some simple code to solve differential-algebraic equations (DAEs). These are equations that have algebraic relationships between variables that are in turn used as the right-hand sides of other variables. They are commonly used in chemical reactor theory. A typical DAE has the form:

$$\frac{du}{dt} = f(u,v) \quad 0 = g(u,v).$$

Programs such as DASSL (Brenan, et al, 1989) and MANPAK (Rheinboldt) handle equations like this fairly easily but use very different methods. *XPPAUT* uses a rather simple scheme which is related to the methods used in MANPAK but not nearly as sophisticated. Newton's method is used to solve the algebraic part and then the ODE part is solved using the results from the Newton method. For example consider this trivial example:

$$\frac{dx}{dt} = y \quad 0 = y + x - 1 \quad x(0) = 0, y(0) = 1.$$

The solution to this is $(x,y) = (1-\exp(-t), \exp(-t))$. In *XPPAUT* , we would write the following:

```
# dae1.ode
# simple DAE
x'=y
0= y+x-1
solve y=1
init x=0
aux yy=y
done
```

NOTE. The line `solve y=1` tells *XPPAUT* which variable to solve for and also provides an initial guess for Newton's method. I have added the line `aux yy=y` so that I can also plot the solution y. In *XPPAUT* , these are kept as internal variables so that you must make them plottable. Run *XPPAUT* with this file. Integrate it and check to see if it is close to the actual solution.

This next example is more complicated:

```
# dae2.ode
# shows dramatic failure until the tolerances are tightened
x'=y
0= 0.001*(.5*(y^3+y)+x-1)
solve y=1
init x=0
aux yy=y
done
```

Now, to do this analytically, we would have to solve the cubic for y. This would result in a horrible formula. Let *XPPAUT* do the work. Run this and you should see that *XPPAUT* fails dramatically. This is because Newton's method is not finding the roots with as much accuracy as you might want. The superfluous small number 0.001 multiplying the relationship between y and x is making the Newton solver think it has found a root! This can be easily remedied by tightening the tolerance of the Newton solver. Click on `nUmerics sIng pt ctl` (**U I**) and change the Newton tolerance from 0.001 to 1e-8. Exit the numerics menu and integrate the equations again. This is much more reasonable. The main point of this little exercise is that you cannot always believe the numerics and if there is a surprising result, then you must take every precaution to be sure that it is not an artifact.

Sometimes you have a fully nonlinear equation of the form:

$$F(\frac{dx}{dt}, x) = 0.$$

Rewrite this as

$$\frac{dx}{dt} = y \quad 0 = F(y, x)$$

Consider a simple model for an artificial liver device. The model is a one-dimensional PDE. The steady state is thus two-point boundary value problem:

$$0 = u_{xx} - vu_x + \lambda_u(w - u) \quad 0 = -k\frac{w}{1 + w} + \lambda_v(u - w).$$

with boundary conditions

$$u(0) = u_0 \quad u_x(L) = 0.$$

We could solve this for w and the result is a rather cumbersome expression. Instead, we let *XPPAUT* do the work. We choose $L = 5$ in this example. We first write this as a system of first order equations. Here is the file for the BVP:

```
# dae_ex3.ode
#  a boundary-value DAE model
u'=up
up'=v*up+lamu*(u-w)
# DAE stuff
0= -k*w/(1+w)+lamv*(u-w)
solve w=.235
# some extra stuff
init u=1,up=-.34
aux ww=w
aux z=-k*w/(1+w)+lamv*(u-w)
par k=1,lamu=.25,lamv=.25,v=.2,u0=1
bdry u-u0
bdry up'
@ total=5.001,jac_eps=1e-5,newt_tol=1e-5,dt=.005
done
```

To solve this, click on `Bndryval Show` and watch it converge to a solution.

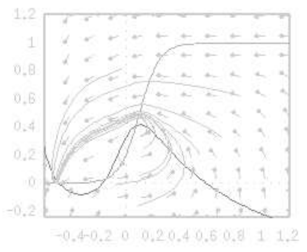
5.3.1 Exercises

1. In this example, you can watch how the DAE solver fails as it reaches a turning point. The equation is
$$\left(\frac{dx}{dt}\right)^2 + x^2 = 1$$
with initial condition, $x(0) = 0$ and $x'(0) = 1$. Write an ODE file for this and try to solve it using *XPPAUT*. At about $t = 1.5$ the solver will fail since $v = dx/dt$ will be close to zero and so $\partial g(x, v)/\partial v$ will be small. Newton's method will be unable to converge. (Note when the DAE solver fails and you want to change some numerical tolerances to perhaps make it work better, you will often have to reset the starting guess. Click on `Initialconds DAE guess` to reset the initial guess.) There is no cure for this failure; a solver such as MANPAK has additional methods to deal with rank 1 Jacobian matrices.

2. In this example, we set the tolerances to be rather crude allowing for a jump to a new branch in order to solve a classic relaxation oscillator. The van der Pol oscillator has the following form:
$$\epsilon\frac{dV}{dt} = v - v^3 - w \quad \frac{dw}{dt} = v.$$
In the limit as $\epsilon \to 0$ it becomes a DAE
$$0 = v - v^3 - w \quad \frac{dw}{dt} = v.$$
The problem is that for $w \in (-0.38, 0.38)$ there are three roots to this equation. The "outer" branches of the cubic are the relevant roots and when w reaches a maximum of the cubic, V must jump to the other root. With close tolerances, the DAE solver will run into the same problems as in problem 1. However, loosening the tolerances a bit will allow the Newton solver to find the other root and make the "jump." Try this ODE file

```
#dae_ex5.ode
# van der Pol with epsilon=0
w'=v_
0= v_*(1-v_*v_)-w
solv v_=1
aux v=v_
init w=0
@ NEWT_ITER=1000,NEWT_TOL=1e-3,JAC_EPS=1e-5,METH=qualrk
@ total=10,dt=.01
@ xlo=-1.5,xhi=1.5,ylo=-.5,yhi=.5,xp=v,yp=w
done
```

Look at the data in the browser and verify that the jump in V is discontinuous.



Chapter 6

Spatial problems, PDEs, and boundary value problems

In space no one can hear you scream.
–Ad for the movie *Alien*

Spatially distributed models are common in physics, biology, and chemistry. The numerical solution of one-dimensional problems is possible to a limited extent using *XPPAUT*. You first have to discretize the system to convert it to a system of ODEs. Then you can take advantage of *XPPAUT* 's numerical and graphics capabilities to solve the resulting systems. Often, you are only interested in finding steady state solutions to a partial-differential equation (PDE). In this case, the resulting equations are sometimes an ordinary differential equation with special boundary conditions. This leads to a boundary value problem; *XPPAUT* and AUTO are both able to solve complex boundary value problems.

6.1 Boundary value problems.

Boundary value problems (hereafter BVPs) present a more difficult problem both numerically and theoretically than do the sorts of initial value problems that we have already described. In BVPs, conditions are given at both ends of the time interval. Often this interval is infinite which confounds things even more. In general, BVPs arise as steady states to certain partial differential equations. The boundary conditions for the PDEs lead to boundary conditions for the ODEs that are derived from them. It is not always the case that a solution even exists. Indeed, unlike the situation with initial value problems, even the smoothest BVPs may not have a solution. Sometimes, they have a solution only when there are some particular parameters chosen correctly. Take the classic problem of a vibrating string:

$$u_{tt} = u_{xx}$$

with $u(0,t) = u(1,t) = 0$. One looks for periodic solutions of the form $u(x,t) = \exp(i\omega t)v(x)$. Clearly,

$$-\omega^2 v = v_{xx}.$$

This is an eigenvalue problem where ω is the unknown parameter. We know that the solution to this is $v(x) = \sin(n\pi x)$ and $\omega = \omega_n = n\pi$. How could we solve this numerically? We note that $v = 0$ is always a solution for any ω so we want to avoid this. Since the problem is linear and second order, we can assume that $v_x(0)$ is not zero (if it were, then $v = 0$ by the uniqueness theorem for ODEs.) So let's, set $v_x(0) = 1$. Then we have a second order equation with three conditions, $v(0) = 0, v_x(0) = 1$ and $v(1) = 0$. There are three "free" parameters, the two initial conditions and the eigenvalue. Thus, there is hope that we can solve this equation.

XPPAUT solves boundary value problems using a very simple algorithm called shooting. For each free parameter there should be a differential equation so that the number of differential equations equals the number of conditions that must be satisfied. The eigenvalue problem has only two differential equations. However, we can add the trivial differential equation:

$$\omega_x = 0$$

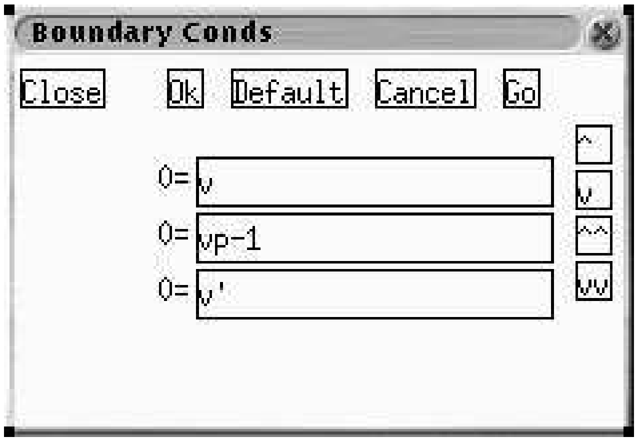
which just says that ω is a constant. Now given that there are N differential equations and N conditions, how does *XPPAUT* solve the equations. Suppose that $\Phi(t; X_0)$ is the solution to the differential equation:

$$X' = F(X,t) \quad X(0) = X_0$$

We have k conditions at $t = 0$ and $N-k$ conditions at $t = L$ the end of the interval. Thus, there are $N - k$ initial conditions that are free to be chosen. This means that we have to solve a subset of $N - k$ nonlinear equations from the conditions $\Phi(L; X_0) = X_L$. The only general way to solve nonlinear equations is Newton's method. This is exactly what *XPPAUT* does. Here is the algorithm:

1. Guess a set of initial conditions. Solve over the given interval. Compute the error. If it is small enough, then exit with success.
2. Numerically compute the Jacobian from the differential equations by making small perturbations of the initial conditions.
3. Use the error in step 1 and the Jacobian in step 2 to improve the initial guess via a Newton iterate:
$$X_0^{new} = X_0^{old} - J^{-1}E$$
where E is the error from step 1.
4. If the Jacobi matrix is uninvertible or too many iterates have been run without convergence, then exit with failure.

Boundary conditions in *XPPAUT* are input in the ODE file or in the **Boundary Cnd window** (shown in Figure 6.1 and obtained by clicking on the `BCs` button along the top of the **Main Window**. Since *XPPAUT* only pays attention to conditions at the endpoints of the integration, you can only have conditions that are given at $t = t_0$ and $t = t_1$ where $[t_0, t_1]$ are the endpoints. The conditions are given



```
0= v
0= vp-1
0= v'
```

Figure 6.1. *The boundary condition window.*

as a series of quantities that are set to zero. Thus, if you want the variable, u, to be -1 at $t = t_0$ the condition you would input is $u+1$. The variables are given different names at $t = t_1$; they are primed. So that if you wanted $u(t_0) = -1$ and $u(t_1) = 1$ you would type the conditions in the **Boundary Cnd window**.

$$0 = u + 1$$
$$0 = u' - 1.$$

The last condition tells *XPPAUT* to try to make $u = 1$ at $t = t_1$. The syntax for these conditions in the ODE file is

```
bndry u+1
bndry u'-1
```

(Note you do not have to write `bndry`, rather `b` alone will suffice.)

With these preliminary comments, here is the ODE file, `eigen.ode` for our eigenvalue problem:

```
# standard eigenvalue problem
# omega is unknown
v'=vp
vp'=-omega*omega*v
omega'=0
b v
b vp-1
b v'
init v=0,vp=1,omega=1
@ dt=.002,total=1.0
@ xhi=1
done
```

We note the following: (i) we rewrite the second order equation as a pair of first order equations; (ii) we have added an additional differential equation for the free parameter, ω; (iii) the boundary conditions attempt to make $v(0) = 0$ and $dv/dx(0) = 1$ and $v(1) = 0$; (iv) I set the known initial conditions to the proper values and guess the unknown `omega=1`; and (v) I have set the total amount of time to 1.00

6.1.1 Solving the BVP

Start *XPPAUT* with this file. Try integrating it. Notice that the solution does not come close to $v = 0$ at $t = 1$. Click on `Bndryval Show` . You will see a few trajectories flash by and then the last one is shown. It is the computed solution. Look in the initial condition window and see that ω has been computed to be 3.141592648 which is very close to the true solution of π. Try a guess of $\omega = 5$ and see what happens. You will get an eigenvalue close to 2π and the second harmonic. Try guessing $\omega = 35$ and you will get an eigenvalue close to 11π. There are other options in the `Bndryval` command. The option you have chosen draws intermediate guesses while the option `No show` only draws the last one.

As we noted above, not every boundary value problem has a solution. Consider the equation

$$u' = -ku \quad u(0) = 1 \quad u(1) = 0.$$

There is no solution to this. However, run the ODE file

```
u'=k*u
k'=0
b u-1
b u'
init u=1,k=1
@ dt=.01,total=1.000
@ xhi=1
done
```

and you will find that *XPPAUT* "finds" a solution in which k is -12 or so. You will notice that $u(1) = 6 \cdot 10^{-6}$ which is about $\exp(-12)$. As far as *XPPAUT* is concerned, it has "found" a solution within the tolerances set for the BVP solver. (Newton's method doesn't really find a zero to a function but only approximates it to within a certain tolerance.) Click on `nUmerics bndVal` (**U V**) and change the tolerance to `1e-10`. Now solve the BVP again (using the value $k = -12$ from before as the initial point). Notice that it again finds a solution but $k = -24$. This is a significant difference from $k = -12$ which should send up red flags that there is an inability to converge to a true solution.

Dendrites are branches that occur on neurons and act as the main input regions for the neurons. In the classical theory of dendrites, they are modeled as passive continuous multi-branched cables. The steady state voltage distribution on such cables is often of interest in simplified models. The distribution of voltages depends on the boundary conditions at the ends of the finite cable. There are several boundary conditions that are relevant: (i) fixed, in which the potential is held constant at one end; (ii) sealed, in which the gradient of the potential is zero; and (iii) leaky, in which there are conditions on the voltage and it's derivative. Consider a case where the dendrite is held at $V = V_0$ at $x = 0$ and has a leaky boundary condition at $x = 1$. Then the equations are:

$$\begin{aligned} \lambda^2 \frac{d^2V}{dx^2} &= V(x) \\ V(0) &= V0 \\ aV(1) + b\frac{dV(1)}{dx} &= 0 \end{aligned}$$

The parameter λ is the space constant for the dendrite. When $b = 0, a \neq 0$ the voltage at $x = 1$ is held at 0. When $b \neq 0, a = 0$ there is no leak from the cable and the conditions are for sealed end. Since this is a second order equation and *XPPAUT* can only handle first order, we write it as a system of two first order equations in the file which is:

```
# Linear Cable Model cable.ode

# define the 4 parameters, a,b,v0,lambda^2
par a=1,b=0,v0=1,lam2=1
# equations
v'=vx
vx'=v/lam2

# The initial data
v(0)=1
vx(0)=0

# and finally, boundary conditions
# First we want V(0)-V0=0
bndry v-v0
#
# We also want aV(1)+bVX(1)=0
bndry a*v'+b*vx'
@ xlo=0,xhi=1,ylo=0,yhi=1.2,total=1
done
```

Note that I have initialized V to be 1 which is the appropriate value to agree with the first boundary condition. With the choice of $a = 1$ and $b = 0$ the end of the dendrite is pinned to 0.

Run this and you will see that the solution grows exponentially and is not even close to the correct value. Click on `Bndryval Show` (**B S**) and the solution will be computed and drawn. (Note that this is a linear equation, so Newton's method will get the right answer after one iteration.) Change $b = 1$ and $a = 0$ and rerun. Notice that big difference in the resulting steady state potential. Try the following. Set $a = 1$ and $b = 0$. Click on `Bndryval Range` to vary over a range of parameters. Choose `b` as the parameter to vary, start at 0, end at 10, and use 20 steps. Set the `Cycle color` value to 1. You will get a range of curves with different colors converging close to the sealed end curve. (Note that these are not permanently stored in memory. You could turn on the Freeze curve option and run each of these individually.)

Sometimes it is necessary to resort to tricks in order to numerically solve a BVP. For example, in some BVPs that arise from partial-differential equations in the unit ball or disk, there are singularities at the origin. Thus, to solve them numerically, you must often move away from the origin. This can be done by computing a local series expansion near the origin. Such is the case in the following problem which arises in the search for rotating waves in a two-dimensional oscillatory medium. The equations for the PDE are

$$\frac{\partial z}{\partial t} = z(1-(1-iq)z\bar{z}) + d\nabla^2 x$$

which is the normal form for a Hopf bifurcation in a two-dimensional diffusive medium. The equations are defined on a unit disk with Neumann boundary conditions. We look for rotating waves in this equation that take the form:

$$z(x,y,t) = A(r)\exp[i(\Omega t - \theta + \int_0^r k(s)\,ds)].$$

Here r, θ are the usual polar coordinates. Substituting this form into the equations reduces the existence of rotating waves in the unit disk to a BVP problem:

$$\begin{aligned} 0 &= A(1-A^2) + d(A'' - (A/r)' - Ak^2) \\ \Omega &= qA^2 + d(k' + k/r - 2kA'/A), \end{aligned}$$

with boundary conditions, $A'(1) = k(1) = 0$, and

$$\lim_{r\to 0}\frac{A(r)}{r} = A'(0) < \infty \qquad \lim_{r\to 0}\frac{k(r)}{r} < \infty.$$

A Taylor series expansion near the origin reveals that, in fact, $k(r)/r \to 0$ as $r \to 0$. This is a third order boundary value problem but there are 4 conditions that must be met. However, the parameter Ω is "free" so that we actually have four degrees of freedom; there is hope. In Paullet et al the existence of solutions to this problem was proven for sufficiently small values of q, d however, the form of the solutions and the extent of the existence is not evident from the proof. In order to solve this using *XPPAUT* , we want to cast it as a system of first order differential equations. Since we are not starting at the origin, it is best to introduce an equation for the independent variable r. The following ODE file will do the trick:

```
# greenberg problem set up for [r0,1+r0]
# gberg_auto.ode
#
```

```
init a=0.00118  ap=1.18  k=0  omeg=-0.19,r=.001
# domain is reasonably small sqrt(1/d)
par d=0.177  q=0.5  sig=3,r0=.001
# the odes...
a'=ap
ap'=a*k*k-ap/r+a/(r*r)-a*(1-a*a)/d
k'=-k/r-2*k*ap/a-(omeg+q*a*a)/d
# extras to make it autonomous
omeg'=0
r'=1
# the boundary conditions
# at r=0
bndry  a-r0*ap
bndry  k
bndry r-r0
# at r=1
bndry  ap'
bndry  k'
# set it up for the user
@ xhi=1.001,dt=.01,total=1.001,ylo=0
done
```

The file is pretty self-explanatory. Note that I have included a differential equation for r the radial coordinate which is the independent variable. The solution to this is $r = r_0 + t$ so that when $t = 0$, $r = r_0$. We choose r_0 small and positive to avoid division by zero. The boundary condition for r at 0 forces $r = r_0$. The boundary condition, `bndry a-r0*ap` forces $a(r_0) = r_0 da/dr(r_0)$ which is only approximate but the error is small as $r_0 \to 0$. Finally note, that the interval of integration here is $[r_0, 1 + r_0]$ which is a bit longer than 1, but doesn't matter since the diffusion coefficient, d can always be rescaled.

Run *XPPAUT* on this file and solve the boundary value problem. (Click `Bndryval` `No show`) and it should draw a nice solution. Change the plot variable to k by clicking `Viewaxes` `2D` and change the `Y-axis` to `k`. Now let's look at the solution over a range of values of q. Click on `Bndryval` `Range` which will solve the boundary value problem as a parameter ranges over some values. Fill it in as follows:

Range over: q
Steps: 10
Start: 0
End: 5
Cycle color(Y/N): Y
Side(0/1): 0
Movie(Y/N): N

Then click on **Ok** . You will see a rainbow of parabolas that are the solutions to the boundary value problem. More examples are discussed in Chapter 7

6.1.2 Infinite domains

Boundary value problems on infinite domains are not *generally* possible to do in *XPPAUT*. However some types of problems can be numerically solved by putting them on a large finite domain or by shooting from a fixed point. Consider the differential equation

$$\frac{dX}{dt} = F(X).$$

One of the common types of solution that are sought for this are trajectories joining fixed points. That is, one is interested in a solution that satisfies

$$\lim_{t\to\pm\infty} X(t) = \bar{X}^{\pm}$$

where $F(\bar{X}^{\pm}) = 0$ are fixed points. If $\bar{X}^{+} = \bar{X}^{-}$ the orbit is called a homoclinic trajectory; otherwise it is called a heteroclinic trajectory. Given a fixed point $\bar{X}$ to a system of ODEs, one computes the linear stability for it. Suppose none of the eigenvalues lie on the imaginary axis, k eigenvalues have positive real parts and $n-k$ have negative real parts. Then under fairly general circumstances, there are sets $\Lambda^{\pm}$ with dimensions k and $n-k$ respectively such that every point on Λ^{+} tends to the fixed point as $t \to -\infty$ and every point on Λ^{-} tends to the fixed point as $t \to \infty$. They are called, respectively, the **unstable** and **stable manifolds** for the fixed point $\bar{X}$. Thus, a heteroclinic trajectory going from $\bar{X}_1$ to $\bar{X}_2$ can be regarded as an intersection of the unstable manifold of $\bar{X}_1$ and the stable manifold of $\bar{X}_2$. Now, when can we expect such an intersection to occur? If the sum of the dimensions exceeds the dimension of the total system, then it the intersection of these is “generic”; that is, it is not surprising. For example, suppose that $n = 1$ and there are two fixed points, stable and unstable. Then we expect that there is a solution joining them. Suppose that $n = 2$, one fixed point is a saddle point and the other is a stable node. Then the existence of trajectory joining the two fixed points is generic. However, if the sum of the the dimensions is less than or equal to the dimension of the system, then we cannot expect this to generically happen. For example, suppose that you want to find a homoclinic in two dimensions. Then the one-dimensional unstable manifold must intersect the one-dimensional stable manifold of the fixed point. This would not be expected unless there was some free parameter, λ. This is illustrated in figure 6.2. As some parameter changes, the unstable and stable manifolds “switch” positions. At a critical value of the parameter $\lambda = 0$, they intersect resulting in the homoclinic orbit.

XPPAUT allows you to compute **one**-dimensional stable and unstable manifolds to an equilibrium point for an ordinary differential equation. This is done by linearizing about the fixed point and then computing the eigenvector associated with the single real positive or negative eigenvalue. This eigenvector is tangent to the manifold, so that it serves as an approximation. If the manifold is the stable manifold, then the system is integrated backwards in time to take it away from the fixed point. If the manifold is an unstable one, then the system is integrated forward in time. Many problems such as traveling waves to reaction-diffusion equations lead to systems in which there is a one-dimensional invariant manifold, so

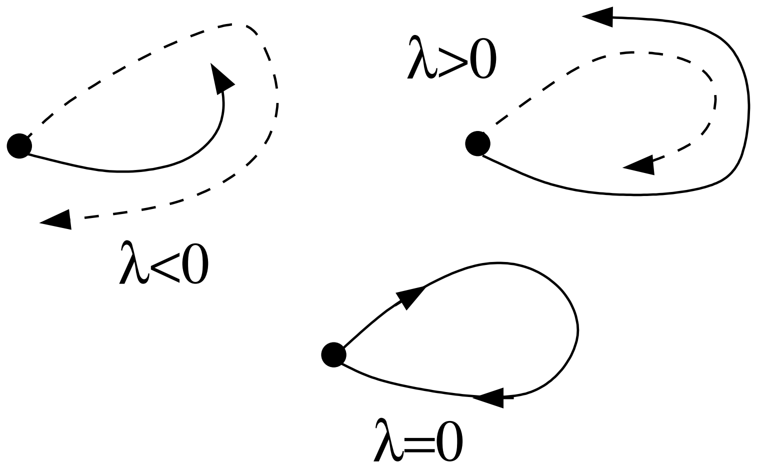


Figure 6.2. *The connection of the unstable (dashed) and stable (solid) manifolds of a fixed point as the parameter, λ varies.*

this restriction to one-dimension is not as bad as it seems. In the next chapter, we show how to find more general homoclinic orbits. Let's turn to several increasingly difficult examples. We will start with the standard example, the Fisher equation:

$$u_t = u_{xx} + u(1-u),$$

which is a model for the spread of an advantageous gene. There are two constant steady states, $u = 0, 1$. We are interested in a constant speed traveling wave front that joins the unstable state 0 to the stable state 1. We look for solutions of the form $u(x,t) = U(x - ct)$ leading to the differential equation:

$$-cU' = U'' + U(1-U).$$

This can be written as a system:

$$U' = V \quad V' = -cV - U(1-U).$$

There are two fixed points, $(0,0)$ and $(1,0)$. Linearizing about these fixed points, we see that $(0,0)$ has eigenvalues $-c/2 \pm \sqrt{c^2-4}/2$ so that this is either a stable node $(c > 2)$ or a spiral $(0 < c < 2)$. Thus, the stable manifold is 2-dimensional. The fixed point at $(1,0)$ is a saddle-point; there is a one-dimensional stable manifold and a one-dimensional unstable manifold. Since the traveling coordinate is $x - ct$ and $c > 0$ we seek a wave going from $U = 1$ to $U = 0$ so that we want to follow the unstable manifold of the point $(1,0)$ into the point $(0,0)$. Since the stable manifold of $(0,0)$ is two-dimensional we expect that this will happen for a whole interval of values of c. The only constraint is to recall that U is a population, so that it can never fall below 0. Thus, the trajectory into $(0,0)$ must be monotone. Since $c < 2$ implies that the origin is a spiral, this means that we better take $c \geq 2$.

Let's see how to use *XPPAUT* to compute these trajectories. As we said above, *XPPAUT* lets you compute one-dimensional invariant manifolds, so we will work from the fixed point $(1,0)$ which has a one-dimensional unstable manifold. Here is the ODE file

```
# fisher.ode
# traveling waves in fisher's equation
#  u_t = u_xx + u(1-u)
#
# -cu' = u'' + u(1-u)
#
f(u)=u*(1-u)
u'=v
v'=-c*v-f(u)
par c=2
init u=1
@ xp=u,yp=v,xlo=-.25,xhi=1.25,ylo=-.5,yhi=.5
done
```

I have set it up so that the phaseplane is the default plot. I have set the velocity to the critical value, $c = 2$.

How to shoot from a fixed point. Here are the steps to "shoot" for a good traveling wave.

1. Set the initial conditions near (or at) the fixed point from which you want to shoot.

2. Click on `Sing pts Go` (**S G**).

3. Answer `No` to the `Print eigenvalues` question.

4. Answer `Yes` to the `Compute invariant sets` question.

5. Click **Esc** when the message boxes come up or when the computation of the trajectory seems to have have converged to a fixed point.

XPPAUT integrates the equations for arbitrarily long times, so it is often necessary to stop the computation when the solution runs into a fixed point. Try this with the Fisher equation. Click on `Sing pts Go` . Answer no to the `Print eigenvalues?` question. At this point a small triangle will appear at the point $(1,0)$ indicating a saddle-point. A new window will appear that summarizes the information about the fixed point. The value of the fixed point is given, the number of real and complex eigenvalues with positive and negative real parts, and a statement about the stability of the fixed point. (E.g., `r+=1` means that there is one positive real eigenvalue and `c-=0` means that there are no complex eigenvalues with negative real parts.) Recall that the eigenvalues of the linearized system inform you of the local stability of that fixed point. *XPPAUT* computes the Jacobi matrix numerically and then uses standard eigenvalue routines to compute the eigenvalues of the resulting matrix. Answer yes to the invariant sets question. *XPPAUT* will first compute unstable and then stable manifolds. The first trajectory computed (yellow) goes

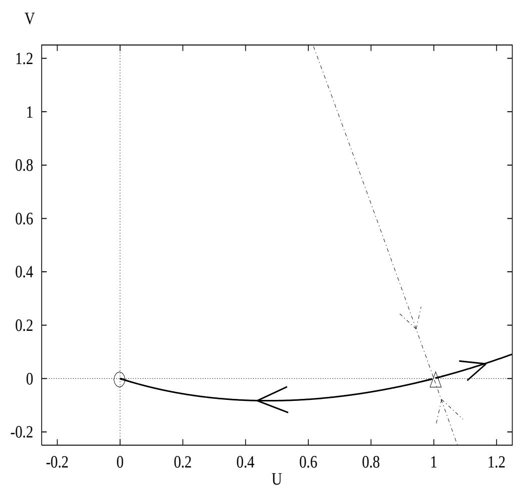


Figure 6.3. *Stable and unstable manifolds for the fixed point (1,0) of the Fisher equation.*

off to the right and out of bounds. The second trajectory goes to the left and enters the fixed point $(0, 0)$ where it is stuck until you click **Esc** . Next, two more (blue) trajectories are computed and these go out of bounds. In *XPPAUT* , the unstable manifolds are drawn in yellow and the stable in blue. If you don't like this choice, you can change it (see Appendix B). The second yellow trajectory is the one we are interested in; it is the branch of the unstable manifold that joins with the fixed point (0,0). Change c to 1 and repeat the calculation. Note that the solution trajectory crosses $U = 0$ and spirals into $(0, 0)$. Change c to 3 and repeat again. This is a perfectly valid solution. Sometimes the initial data are quite far from the fixed point. This means that the approximation to the invariant manifold is probably pretty bad. To make the initial data closer, make `dt` smaller within the `nUmerics` menu.

One point to note is that *XPPAUT* does not keep these trajectories in memory so that you cannot get hard copy of them. However, *XPPAUT* is aware of the initial conditions, so you can repeat the calculations using the appropriate conditions. For example, we want the **second** trajectory computed. This is an unstable manifold so if we compute forward in time, we will go away from the fixed point as desired. Click on `Initialconds s(H)oot` (**I H**) and choose **2** when given the choice of which since we want the **second** trajectory that *XPPAUT* computed. You will get about two thirds of the trajectory. You should extend the amount of time to integrate to say 40 to get the full trajectory. For now, just click on `Continue` (**C**) and continue to 40. You should have the traveling front. To see the profile of the front, plot U against t. (Tap **X** and choose `U` instead of `V` and press **Enter** .) Figure 6.3 shows a complete picture of the unstable and stable manifolds for the fixed point (1,0) including the trajectory that represents the traveling front. Arrows indicate the directions; these are added using the `Text Arrow` command (see Chapter 9).

The bistable reaction-diffusion equation. The Fisher equation is in a sense rather trivial as the wave exists over a whole range of parameters. In most systems, you must choose the velocity exactly right. This requires that you repeat the steps above many times as you attempt to home in on the correct parameter. The key is that you want to find a parameter that is below the correct one and one that is above the correct one. How can you tell? In most one-parameter shooting problems, choosing the parameter low will lead to the trajectory going in one way and choosing it high will lead to it going in another way. Thus, you can usually decide if you are too high or too low. I will illustrate this with the bistable reaction-diffusion equation:

$$u_t = u_{xx} + f(u) \tag{6.1}$$

where $f(u)$ has three roots, $0 < a < 1$ with $f'(0) < 0$, $f'(1) < 0$, and $f'(a) > 0$. Fife and McLeod proved the existence, uniqueness, and stability for traveling front solutions to this equation where, $u(x,t) = U(x - ct)$, $U(-\infty) = 1$ and $U(\infty) = 0$. That is, there is a unique value of the velocity c such that this equation has a traveling wave solution. The associated ODE is

$$-cU' = U'' + f(U).$$

We will shoot from $U = 1, U' = 0$ forward in time and attempt to hit $U = 0, U' = 0$. A linearization reveals that both points are saddle-points. The unstable manifold of (1,0) and the stable manifold of (0,0) are both one-dimensional, so we expect that there will not be a solution unless c is chosen appropriately. Here is the ODE file for this problem:

```
# bistable.ode
# classic example of a wave joining 0 and 1
#  u_t = u_xx + u(1-u)(u-a)
#
# -cu'=u''+u(1-u)(u-a)
f(u)=u*(1-u)*(u-a)
par c=0,a=.25
u'=up
up'=-c*up-f(u)
init u=1
@ xp=u,yp=up,xlo=-.5,xhi=1.5,ylo=-.5,yhi=.5
done
```

We have written the ODE as a pair of first order equations. The initial data is set at $(1, 0)$ so that we want the unstable manifold of this saddle point to hit the point $(0, 0)$. The window is set up as the phase-plane with an appropriate view. Now we let *XPPAUT* compute the one-dimensional invariant manifolds. Click on `Sing Pts Go` (**S G**) and *XPPAUT* will quickly ask if you want the eigenvalues printed. Click on **No**. Next you will be prompted as to whether you want to compute the invariant sets. Since that is the whole goal of this exercise, we will answer yes. You should see a trajectory drawn off to the top right and you will get an out of bounds message. Click **Ok** and the next invariant set will be computed. Again click **Ok** and continue in this way until all four invariant sets are computed. The first two trajectories computed represent the unstable manifolds of the fixed point. (How do I know this? *XPPAUT* draws the stable (unstable) manifolds in blue (yellow) and always computes the unstable branches first.)

Use the `Initialconds sHoot` command to recreate these trajectories if you want – integrate **forward** in time for unstable and **backward** for stable manifolds. To integrate **backward**, click on `nUmerics Dt` and change the integration stepsize from positive to negative. When you choose `Initialconds sHoot` , you are given 4 choices; pick 2 as the second trajectory is the branch of the unstable manifold desired. Freeze this curve in memory (`Graphics Freeze Freeze`) and accept the defaults.

Now, change c to 0.5. Once again, click on **S G** to compute the fixed point; answer **No** to the `Print eigenvalues` prompt, and **Yes** to the `Compute invariant sets` prompt. As before, the first trajectory becomes unbounded and requires a response. But the second trajectory does not go off to infinity. Rather, is is attracted to the fixed point at $(a, 0)$. *XPPAUT* always integrates until a solution goes out of bounds when invariant sets are computed. Since this solution will never go out of bounds, you must interrupt the integration by tapping **Esc** , the escape key. Finish the computation of the stable manifolds. As before, you can compute this new trajectory using the `Initialconds sHoot` command and choosing solution #2. Freeze this one too. (See Figure 6.4.)

We now have defined our "shooting sets." For c close to zero, the unstable manifold hits the u' axis before it hits the u axis. For c large enough, the unstable manifold hits the u axis before the u' axis. By continuity with respect to parameters, there must be a value of c between 0 and 0.5 for which the stable manifold crosses both axes at the same time; that is, it hits $(0, 0)$, our shooting target. (One must avoid the cases for which the trajectory is tangent to the axes. Therein lies the hard analysis required to make this argument into a rigorous proof.) So, the procedure is clear. Refine your choice of c according to which of these two conditions hold. If the solution hits the u' axis, increase c and if it hits the u axis, decrease it.

Try $c = 0.25$. Notice that it hits the u' axis, so it is too small. Choose a point halfway between 0.25 and 0.5, $c = 0.375$ and notice that this time c is too big. Thus, we know that the velocity of the traveling wave is between 0.25 and 0.375. Again choose c halfway between these values, $c = 0.3125$ and run again. It is too small! Make it bigger, say 0.34375. This turns out to be too small also. Several more guesses narrow the values of c to between 0.34375 and 0.359375. Figure 6.4 shows three trajectories; one of which is very close to the correct value of c.

6.1.3 Exercises

1. Reconsider the eigenvalue problem:

$$u_{xx} = -\omega^2 u.$$

The way that we avoided the trivial solution, $u(x) = 0$ was to set the derivative to 1 at $x = 0$. A more standard way of normalizing the solution to the linear

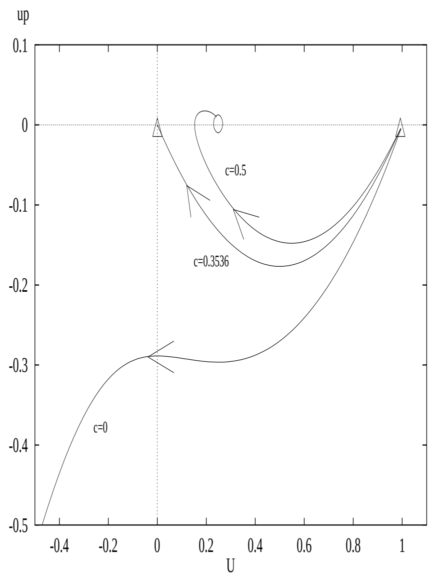


Figure 6.4. *Establishing shooting sets for the bistable reaction-diffusion equation.*

problem is to force it's L_2 norm to be 1:

$$\int_0^1 u^2(x)\,dx = 1.$$

So, how can we do this in *XPPAUT* ? Consider the differential equation:

$$\frac{dz}{dx} = u^2$$

along with the boundary conditions

$$z(0) = 0 \qquad \text{and} \qquad z(1) = 1.$$

Clearly, $u = 0$ is not a solution. Thus, we can write our boundary value problem as this four-dimensional system:

$$\begin{aligned} u' &= v \\ v' &= -\omega^2 u \\ z' &= u^2 \\ \omega' &= 0 \end{aligned}$$

with the four conditions: $u(0) = u(1) = z(0) = 0$ and $z(1) = 1$. Try solving this with *XPPAUT* using different initial guesses for ω.

2. Solve the nonlinear problem

$$u' = v \qquad v' = u(1 - v^2)$$

with the boundary conditions, $u(0) = 0$ and $u(1) = -1/2$.

3. Dendrites with active currents have been the subject of much recent interest in the computational neuroscience community. Consider the model with a nonlinear calcium current:

$$0 = V_{xx} - V + g(V)(E - V)$$

with $g(V) = \bar{g}/(1 + \exp(-(v - 40)/5))$, $E = 200$, and the following boundary conditions, $V(0) = V_{hold}$ and $V_x(5) = 0$. The two parameters of interest are $\bar{g}$ the strength of the calcium conductance, and the holding potential, V_{hold}. Set $\bar{g}$ to zero, $V_{hold} = 100$ and solve the boundary value problem. (Hint start with initial data, $V(0) = 100, V_x(0) = -100$ which is close to the true solution.) Now, you should have a nice solution to the linear problem. We will use the `Bndryval Range` command to compute the solution to the nonlinear BVP as $\bar{g}$ varies between 0 and 0.8. Click on `Bndryval Range` . Choose `gbar` as the parameter and range between 0 and 0.8 with 40 steps. A whole lot of solutions will emerge. The **Data Viewer** is filled with the following information: The first column has the parameter and the remaining columns contain the value of $V(0)$ and $V_x(0)$. Write a few down, noticing that there is an abrupt transition in behavior at about $\bar{g} = 0.55$; to see this plot `v_x` versus `t` which contains the parameter. Figure 6.5 shows a plot of one solution and the behavior conductance varies.

4. A simplified model for excitability in a nerve is the Morris-Lecar system. In this homework exercise, we first compute the velocity of the front in the absence of a recovery variable and then in the next exercise compute the traveling wave for the full system. The equations for the full PDE are

$$\begin{aligned}
\frac{\partial V}{\partial t} &= \frac{\partial^2 V}{\partial x^2} + I - g_L(V - V_L) - g_K w(V - V_K) - g_{Ca} m_\infty(V)(V - V_{Ca} \\
&\equiv \frac{\partial^2 V}{\partial x^2} + f(V, w) \\
\frac{\partial w}{\partial t} &= \phi \lambda_w(V)(w_\infty(V) - w) \\
&\equiv g(V, w).
\end{aligned}$$

The functions $w_\infty(V), m_\infty(V), \lambda_w(V)$ involve exponentials of the voltage and are given in the ODE file for the traveling wave below. In absence of diffusion, there is a unique stable fixed point, (V_r, w_r). One is interested in the existence of traveling pulses, that is solutions that are functions of $x - ct$ and homoclinic to the rest state. A classic way to analyze nerve equations is to assume that the recovery variable, w is slow; that is ϕ is a small parameter. Then one studies the reduced system holding w at its rest value and looks for a front that joins the rest state to the "active" state (in this case, $V = V_{Ca}$). We do this first. Here is the ODE file for the traveling front:

```
# morris-lecar front
# mlfront.ode
param c=0
params v1=-.01,v2=0.15,gca=1.33
params vl=-.5,iapp=.04,gl=.5
minf(v)=.5*(1+tanh((v-v1)/v2))
f=gl*(vl-v)+gca*minf(v)*(1-v)+iapp
v'=vp
vp'=-c*vp-f
init v=1
@ xlo=-1,xhi=1,ylo=-1,yhi=1
@ xp=v,yp=vp
done
```

Here is your first task. Find two values of c such that the unstable manifold from the fixed point $(V, V') = (1, 0)$ either goes off to infinity (c too low) or is pulled into a stable fixed point (c too high). Hint: start with $c = 0$ and $c = 2$. Next, use these two values to successively refine your guess of the velocity until you have it to within two decimal places.

5. We now turn to the full equations. Shooting in this case is more subtle and in fact, there is no mathematical proof of the existence of traveling wave solutions to this particular model. Traveling waves satisfy:

$$-cV' = V'' + f(V, w) \quad -cw' = g(V, w)$$

with $(V(\pm\infty), w(\pm\infty), V'(\pm\infty)) = (V_r, w_r, 0) \approx (-0.4, 0, 0)$. It is easy to show that this equilibrium point has a one-dimensional stable manifold and a two-dimensional unstable manifold. Thus, as with the front, we don't generically expect that there will be an intersection if the one-dimensional manifold with the two-dimensional manifold in the three-dimensional phase space. However, we have the free parameter c to vary. We must establish the correct shooting set. Here is the ODE file, `mlwave.ode`:

```
# morris-lecar traveling wave
param c=.5
params v1=-.01,v2=0.15,v3=0.1,v4=0.145,gca=1.33,phi=.333
params vk=-.7,vl=-.5,iapp=.04,gk=2.0,gl=.5,om=1
minf(v)=.5*(1+tanh((v-v1)/v2))
ninf(v)=.5*(1+tanh((v-v3)/v4))
lamn(v)= phi*cosh((v-v3)/(2*v4))
f(v,w)=gl*(vl-v)+gk*w*(vk-v)+gca*minf(v)*(1-v)+iapp
g(v,w)=lamn(v)*(ninf(v)-w)
v'=vp
vp'=-c*vp-f(v,w)
w'=-g(v,w)/c
init v=-.4,w=0,vp=0
@ xp=v,yp=w,xlo=-.5,xhi=.3,ylo=-.1,yhi=.6
done
```

Note that the one-dimensional manifold is a stable manifold in this case. Since ϕ is reasonably small, we expect that if there is a traveling wave, then it will have a velocity close to that of the front (which is the velocity of the pulse in the limit as $\phi \to 0$). Set the velocity, c to 0.5 and shoot using the `Sing Pts Go` command. Observe what the right branch of the unstable manifold does. Change c to 2 and repeat the calculation. Notice that the branch again goes to infinity but via a different path. Try $c = 1$ and $c = 1.5$. Try to get a value of c that is good to two decimal places. Plot two guesses on the same graph that are on either side of the correct value. (Hint, since this is a *stable* manifold, we must integrate **backwards** to compute it. The second branch is the desired one to compute using the `Initialconds sHoot` command.)

Finally, choose $c = 1.08121222$ and shoot. Then set the total time to 13 and integrate the equations backwards with the appropriate initial condition and plot (v, w) versus t, the independent variable. You will see a good approximation of the pulse over that time scale.

6.2 Partial differential equations and arrays.

XPPAUT has many operators that make it relatively straightforward to solve spatially discretized partial differential equations and a variety of convolution equations. The only downside to this is the somewhat limited size of the arrays that can be solved with *XPPAUT*. (The number of equations *XPPAUT* can integrate in recent versions is about 800.) The task of the user is to determine the appropriate discretization and then code it up. I will start with a simple PDE, a cable of length L with mixed linear boundary conditions at each end. The equations are

$$\begin{aligned}\frac{\partial u}{\partial t} &= D\frac{\partial^2 u}{\partial x^2} - u \\ c_0 &= a_0 u(0,t) + b_0 u_x(0,t) \\ c_L &= a_L u(L,t) + b_L u_x(L,t).\end{aligned} \tag{6.2}$$

The first step is to spatially discretize the PDE to convert it to a set of coupled ODEs. The method of lines is quite suitable so that is what we will use:

$$u_j' = \frac{D}{h^2}(u_{j+1} - 2u_j + u_{j-1}) - u_j \quad j = 1, \ldots N.$$

Here $h = L/N$ is the grid size and N is the number of grid points. The boundary conditions are coded in the equations for the end points, u_0 and u_{N+1}. We can write the boundary condition at $j = 0$ as

$$c_0 = a_0 u_0 + b_0(u_1 - u_0)/h$$

obtaining

$$u_0 = (c_0 - b_0 u_1/h)/(a_0 - b_0/h).$$

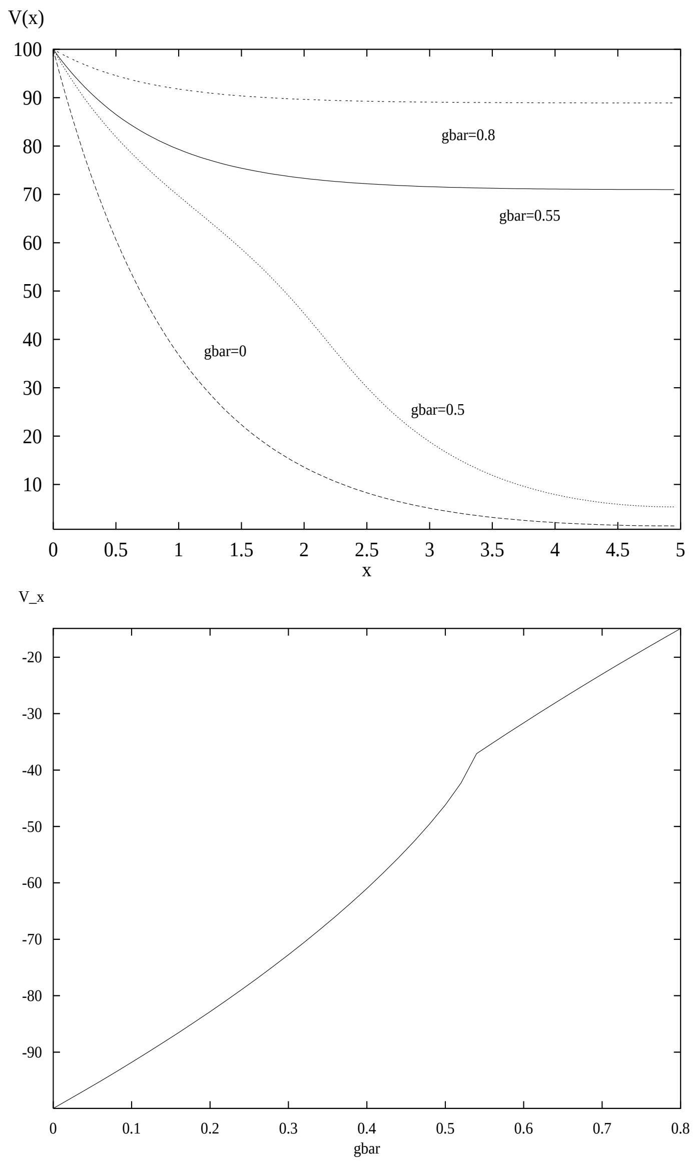


Figure 6.5. *Steady state behavior of a nonlinear dendrite. Top: Some representative solutions. Bottom: Flux at $x = 0$ as a function of $\bar{g}$.*

Similarly, we can solve for u_{N+1} obtaining

$$u_{N+1} = (c_L + b_L u_N/h)/(a_L + b_L/h).$$

Given these two end conditions, all variables in the discretization above are defined at $j = 1$ and $j = N$. Here is the ODE file for a grid of 100 points.

```
# cable100.ode
# cable equation with different BC's
#  c0 = a0 u + b0 u_x
#  cl = al u + bl u_x
#  h=0.1 L=100 h
u0=(c0-b0*u1/h)/(a0-b0/h)
u[1..100]'=(d/(h*h))*(u[j+1]-2*u[j]+u[j-1])-u[j]
u101=(cl+bl*u100/h)/(al+bl/h)
par d=1,h=.1,c0=1,a0=1,b0=0,cl=0,al=0,bl=1
@ total=5,dt=.005
done
```

Arrays are set up by using the `x[n..m]` construction which defines `m-n+1` variables `xn, ..., xm`. Thus in this example, `u50` would be the variable in the middle of the domain. The name of the array variable is `j` and thus, `u[j+1]` defines the variable `u51` if `j=50`. In the present example, we have set the boundary conditions to be $u(0,t) = 1$ and $u_x(L,t) = 0$. The initial data are all zero in this case.

Once one deals with PDEs there are a number of practical issues that have to be considered. First, you certainly do not want to type in all the initial conditions. Second, you may want to look at spatial profiles rather than temporal ones. Third, you may want to look at the entire space-time picture. *XPPAUT* has some commands that make this pretty easy to do. Run *XPPAUT* with this file. Let's put initial data in that has the form $u(x,0) = \cos(\pi x/L)$. Since there are 100 elements, then the initial data is `u_j = cos(pi*j/100).` To input this you use the `Initialconds formUla` (**I U**) command. At the prompt for the variable, type in `u[1..100]` to indicate you want to set initial data for `u1, ..., u100`. Then when asked for the formula, type in `cos(pi*[j]/100)` (**Don't forget to put brackets around j!**) and hit `Enter` . Since there are no other variables whose initial data you need to solve, click on `Enter` again. The equations will be integrated.

Let's get a big picture of the entire space time behavior. To do this, we use the `Array plot` options for *XPPAUT*. Click on `Viewaxes Array` (**V A**) and you will get a dialog box to edit. Fill it in as follows:

`Column 1:` U1
`NCols:` 100
`Row 1:` 0
`NRows:` 201
`RowSkip:` 5
`Zmin:` -1
`Zmax:` 1
`Autoplot(0/1):` 0
`Colskip:` 1

and click on `Ok` . A new window appears with a number of buttons. Click on **Redraw** and you should see a space-time plot of the variable $u(x,t)$. "Space" runs horizontally and "time" runs vertically. Since we have integrated the equations with a time step of `0.005` for `5.0` time units, there are 1000 rows of data. We plot every

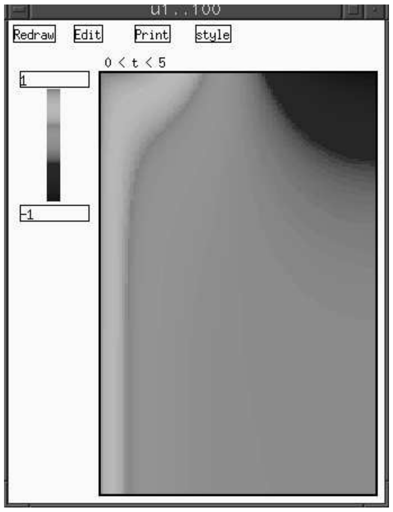


Figure 6.6. *The array plot window showing the space-time behavior of the cable equation.*

5th row. About halfway through the simulation, it appears that the solution has reached a steady state. You should see something like figure 6.6.

(**Cool shortcut:** in the **Initial Data Window** , click on the little box to the left of the variable `U1` and scroll down until you see the variable `U100` and click on its little box. Then click on `arry` in the **Initial Data Window**. This will automatically set up the array for you.)

Let's look at a spatial plot at different times. *XPPAUT* has a "transpose" command which transposes the data in the data browser to allow you to plot the spatial distribution of the variable u. Let's plot $u(x, 0), u(x, .5), u(x, 2.5)$, and $u(x, 5)$. As above, these represent respectively rows 0, 100, 500, and 1000. Click on `File Transpose` (**F T**) and fill in the resulting dialog box as follows:

`Column 1:` U1
`NCols:` 100
`Colskip:` 1
`Row 1:` 0
`NRows:` 11
`RowSkip:` 100

Row 0 will be moved to the column of the first variable, `U1`. Since `Rowskip=100` Row 100 will be moved to the second data column, Row 200 to the third data column, up to Row 1000 in the eleventh data column. The values in the `Time` column will be replaced by the numbers 1 to 100. Thus, if we plot `U1` against `t` we will see the spatial profile of the initial data. Plotting `U2` against `t` will plot the spatial profile of the Row 100 which represents the spatial profile when $t = 0.5$. To get rows 0, 100, 500, and 1000, we thus would plot `U1,U2, U6, U11` respectively. You can do

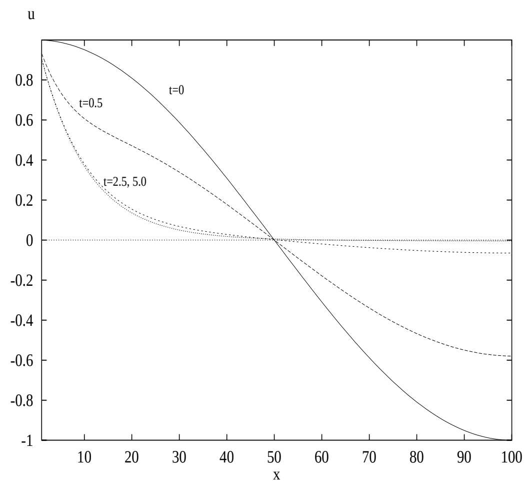


Figure 6.7. *Solutions to the cable equation at 4 different times.*

this manually using the `Graphics` `Add curve` (**G A**) command. Alternatively, here is a shortcut. In the **Initial Data Window** check the boxes corresponding to `U1`,`U2`, `U6`, `U11` and in the **Initial Data Window** , click the button `xvst`. All four curves will appear on the plot. Two of the curves are very close corresponding to $t = 2.5$ and $t = 5$ confirming what we saw in the space-time plot (see Figure 6.7).

6.2.1 An animation file

One more way to look at the evolution of space-time activity is to animate it. *XPPAUT* has some animation routines which we will explore in detail later. Clear all the graphs from the main *XPPAUT* window by clicking `Graphics` `Remove all` (**G R**). Clear the transpose by clicking `File` `Transpose` and click on the **Cancel** button.

Create a file called `cable100.ani` that has the following five lines:

```
# animation for the array
# cable100.ani
vtext .8;.95;t=;t
fcircle [1..100]/100;(u[j]+1)/2.2;.02;[j]/100
end
```

and save it in the same directory as the ODE file. This is an **animation** file and is used to tell the *XPPAUT* animator what to do at each time step. For now, don't worry about the syntax. Click on `Viewaxes` `Toon` (**V T**) and a new window will appear on the screen. This window is the animation window and will have the name of a random cartoon on the title bar. In the animation window, click on the **File** button. Assuming you named the animation file `cable100.ani`, type this in for the file name or select it from the list. Click on the **Ok** button in the animation window.

Assuming there are no errors, you should be able click on the **Go** button and see an animation of the spatial profile as it goes from a cosine wave to a damped exponential.

6.2.2 The issue of stiffness

Discretization of PDE's using the method of lines often leads to sets of differential equations which are numerically stiff. (See the appendix for definitions.) That is, if you use a fixed step integrator, you will have to choose a very small step size in order to avoid numerical problems. If you use one of the standard *XPPAUT* adaptive integrators like `Qualrk` or the Dormand-Prince, integrators, it is also possible to run into problems since these are meant for ODEs which don't have large amplitude right-hand sides. Instead, it is useful to use one of the "stiff" integrators in the *XPPAUT* tool kit. These converge to a solution even if you use a fairly large stepsize. The recommended stiff integrator is `CVODE`.

The disadvantage of stiff integrators is that they must invert a large matrix, possibly many times per time step. By default *XPPAUT* assumes that the matrix that must be inverted is "full". That is, most of the entries are nonzero. However, in PDEs, the equations can often be written so that only a few entries near the main diagonal are nonzero. *XPPAUT* can implement a banded form of CVODE which can be hundreds of times faster!.

For example, consider the reaction-diffusion equation:

$$\begin{aligned} u_t &= D_u u_{xx} + f(u,v) \\ v_t &= D_v v_{xx} + g(u,v). \end{aligned} \tag{6.3}$$

If we discretize this into N points to get $2N$ ODEs, we would have to invert a $2N \times 2N$ matrix. If we defined all the u_j's first and then all the v_j's, then there would be interactions between j, $j \pm 1$ and $j \pm N$. Thus, the resulting matrix is not banded. However, if we organize the discretization as, $u_1, v_1, u_2, v_2, \dots, u_N, v_N$, then the resulting system is banded with a bandwidth of ± 2. *XPPAUT* has a way to write the equations so that they come out in this format. Instead of using the `u[1..100]` syntax, we instead use the `%[1..100]` command which tells *XPPAUT* that everything that follows is an array. Here is the above PDE discretized to 100 points with Neumann boundary conditions, `rd2.ode`:

```
# rd2.ode
# generic 2 variable reaction diffusion with Neumann conds
f(u,v)=u*(1-u)*(u-a)-v
g(u,v)=(u-d*v-b)*c
u1'=f(u1,v1)+du*(u2-u1)/h^2
v1'=g(u1,v1)+dv*(v2-v1)/h^2
%[2..99]
u[j]'=f(u[j],v[j])+du*(u[j+1]+u[j-1]-2*u[j])/h^2
v[j]'=g(u[j],v[j])+dv*(v[j+1]+v[j-1]-2*v[j])/h^2
%
u100'=f(u100,v100)+du*(u99-u100)/h^2
```

```
v100'=g(u100,v100)+dv*(v99-v100)/h^2
par a=.25,d=0,b=.4,c=1,du=.2,dv=1,h=.1
@ meth=cvode,bandup=2,bandlo=2
done
```

Everything between the % signs is considered an array; these equations are expanded as though you had written

```
u2'=f(u2,v2)+du*(u3+u1-2*u2)/h^2
v2'=g(u2,v2)+dv*(v3+v1-2*v2)/h^2
u3'=f(u3,v3)+du*(u4+u2-2*u3)/h^2
v3'=g(u3,v3)+dv*(v4+v3-2*v3)/h^2
...
```

Note how they are interlaced so that the resulting system is nicely banded. I have told *XPPAUT* to use the banded form of CVODE by informing the program about the upper and lower bandwidths.

The problem with this is that when you use the array plot or the transpose, both u and v will be interlaced. The cure for this is to skip every other column in the array plot or transpose. That is, change `Colskip` from 1 to 2!

As an example, consider the complex Ginzburg-Landau equation (CGL):

$$z_t = z - (1 + i\beta)z^2\bar{z} + (1 + i\epsilon)z_{xx}$$

where $z = u + iv$. For certain ranges of the parameters, β and ϵ this model exhibits spatio-temporal chaos. We write it in real form and solve it with a small perturbation of u in the middle of the medium. Here is the ODE file:

```
# coupled Ginsberg-Landau equation
# cgl.ode
!d=1/(h*h)
r[0..127]=u[j]*u[j]+v[j]*v[j]
u0'=u0-(u0-beta*v0)*r0+d*(u1-u0-eps*(v1-v0))
v0'=v0-(v0+beta*u0)*r0+d*(v1-v0+eps*(u1-u0))
%[1..126]
u[j]'=u[j]-(u[j]-beta*v[j])*r[j]+d*(u[j+1]+u[j-1]-2*u[j]-eps*(v[j+1]+v[j-1]-2*v[j]))
v[j]'=v[j]-(v[j]+beta*u[j])*r[j]+d*(v[j+1]+v[j-1]-2*v[j]+eps*(u[j+1]+u[j-1]-2*u[j]))
%
u127'=u127-(u127-beta*v127)*r127+d*(u126-u127-eps*(v126-v127))
v127'=v127-(v127+beta*u127)*r127+d*(v126-v127+eps*(u126-u127))
# parameters
par beta=-1.8,eps=.8,h=1
aux rr[0..127]=r[j]
init u[50..55]=1
@ dt=.25,total=200
# plot the modulus rr0 with 128 columns in the array plot
done
```

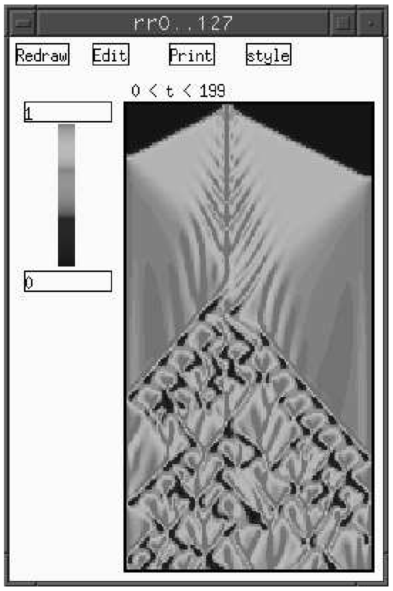


Figure 6.8. *Solutions to the CGL equation.*

I use the default Runge-Kutta integrator since the problem as written is not particularly stiff. The domain size is 128 units and the spatial grid, `h=1`. Two lines bear mention:

```
!d=1/(h*h)
```

defines `d` to be a derived parameter. It is hidden from the user and its value depends on other parameters. It is only evaluated when the parameters are changed. The second notable point is that I have added auxiliary variables for the square magnitude of z. If you make `h` small, you will probably have to switch to a stiff integrator. Run this as is and array plot `rr0`. You will see the interesting pattern shown in Figure 6.8

6.2.3 Special operators

XPPAUT has a number of operators that make it easy to solve discrete convolution equations that arise in physical systems. Here we will numerically solve a neural network model with delayed inhibition. The model has the following form:

$$\frac{du_j}{dt} = -u_j + f[c^+ \sum_{i=0^N} W(j-i)u_i(t) - c^- u_j(t-\tau)]. \tag{6.4}$$

This equation arises as a discrete approximation of a continuous convolution equation, *e.g.*,

$$\int_0^1 W(x-y)u(y)\, dy.$$

We could use the *XPPAUT* sum operator to evaluate this discrete convolution with the code

```
sum(0,N)of(w([j]-i')*shift(u0,i'))
```

but this will be quite slow and would be a real waste if, for example, the function $W(j)$ vanished for $|j|$ large enough. (Recall that `shift(u0,n)` returns the value of the n^{th} variable defined after the variable `u0`.) Suppose that $W(j)$ is nonzero for $|j| \leq m$. Then we could rewrite the code fragment as:

```
sum(-m,m)of(W(i')*shift(u0,[j]+i'))
```

provided we knew how to treat the `shift(u0,i)` when `i` is negative or exceeds the length of the array of variables. *XPPAUT* provides a fast way to evaluate this convolution and also provides several different ways to treat the endpoints. Suppose that we are given an array of variables `u[0..N]` and we want to convolve this with the $2m+1$ weights, $W(-m), \ldots, W(m)$. First we must use the `table` command in the ODE file to create the weight table, W. Then we use the `special` command to create a function whose return value is the desired convolution.

A brief aside on tables In *XPPAUT* you can define functions as lookup tables by using the `table` command. You can either have *XPPAUT* read the values from a file or you can have *XPPAUT* create the table using a function. Suppose you want to create a table of length N for a function, $f(t)$ defined for $t_1 < t < t_2$. That is, the N values of the table are: $f(t_1), f(t_1+dt), \ldots, f(t_2)$ where $dt = (t_2 - t_1)/(N-1)$. Let's call the table `w`, and suppose $N = 51$, $t_1 = -25$ and $t_2 = 25$. Then you write the following in your ODE file:

```
table w % 51 -25 25 f(t)
```

The usefulness of tables is that the function could be very complicated so that there can be a big speed increase by using a lookup table for the function. `w` is treated like a function in *XPPAUT* so that `w(17)` returns `f(17)`. **Note:** the dummy argument for the function is **always** called `t`. For tables defined by functions, linear interpolation is used to obtain values of the function that are not on the grid points of the table. Thus, `w(17.5)` will return $(f(17) + f(18))/2$.

You can also define a table using a data file that has a special structure. To do this, you declare:

```
table w junk.tab
```

where `junk.tab` is a file containing the information for the lookup function `w`. The file has the following format:

```
N
t_1
t_2
f_1
.
```

```
.
.
f_N
```

where `N` is the number of values you will define, `t_1` is the lowest value of domain of the function, `t_2` is the highest value, and `f_1` etc are the values at the `N` discrete points. File-based lookup tables allow you to determine how intermediate values of t are defined. In function-based tables, only linear interpolation is allowed. For file-based tables, there are two other ways to interpolate: piecewise constant and cubic splines. *XPPAUT* determines which is used from the first character in the file. If the first character is 's', then it uses splines; if it is 'i' it uses piecewise constant. Otherwise it uses linear interpolation. Thus for a table with 100 values, the first line of the table will be one of the following three:

```
100
s100
i100
```

and the intermediate values of the function will be defined by respectively, linear interpolation, splines, and piecewise constant. File-base tables are an excellent way to drive a simulation with experimental data.

XPPAUT can produce table files from columns of numbers that are in the data browser. Use the `Read` command in the **Data Viewer** to load the browser with data. Then use the `Table` command in the **Data Viewer** to create a table from the the data in one of the columns.

Back to convolutions. Suppose you have the array `u[0..100]` of 101 elements and you want to convolve it with the weights, `w(-25),...,w(25)`. Then you want to form the sums

$$k(j) = \sum_{l=-25}^{25} w(l)u[j+l].$$

Since u is defined only for 0 to 100, we need a way to handle this sum when $j+l$ is not between 0 and 100. *XPPAUT* provides three possibilities: `zero, even, periodic`. `zero` means that we define u to be zero outside the points 0 to 100; `even` means that the values are reflected across the boundaries. For example, $u[-2] \equiv u[2]$ and $u[105] \equiv u[95]$. Finally, `periodic` means that $u[j \pm N] \equiv u[j]$. The `special` statement in an ODE file allows us to define the function $k(j)$. For this particular example, we would write:

```
table w % 51 -25 25 exp(-abs(t/5))
special k=conv(zero,101,25,w,u0)
```

where we have arbitrarily chosen an exponential decay for the weight function. The `conv` function takes five arguments:

```
special k=conv(type,n,m,w,u0)
```

As described above, `type` is either `even,zero,periodic` depending on how you want to set the edges of the array. The parameter `n` is the length of the array `u[a..b]` and `u0` is the name of the first element in the array. `n` is also the length of the return values for `k`. `m` is the halfwidth of the weight, that is, $w(j)$ is defined for $-m \leq j \leq m$ and `w` is the name of the weight table.

As a warm-up example, we will discretize the piecewise constant wave model

$$\frac{\partial u}{\partial t} = -u(x,t) + \frac{1}{2\sigma}\int_{-\sigma}^{\sigma} H(u(x+y)-\theta)\; dy$$

where $H(u)$ is the Heaviside step function. This nonlocal equation has a traveling front solution. The discrete analogue is

$$u_i' = -u_i + \frac{1}{2m+1}\sum_{-m}^{m} H(u_{j+i}-\theta).$$

We will discretize this into 101 points and choose $m = 5$. We first define a table for the weights:

```
table w % 11 -5 5 1/11
```

This creates a lookup table of length 11 and each entry is just 1/11; a constant weight. Next we create an array in which the step function is applied to u:

```
h[0..100]=heav(u[j]-theta)
```

Thus `h0,h1,` is a set of fixed variables whose values are just the step function applied to the corresponding u variables. Now we define the discrete convolution:

```
special k=conv(zero,101,5,w,h0)
```

This says that we will set the values of of h to be zero outside the domain. Finally, we define the right-hand sides using the special operator `k([j])`:

```
u[0..100]'=-u[j]+k([j])
```

Putting this together yields the ODE file:

```
# convwave.ode
# define a table for the convolution
table w % 11 -5 5 1/11
# define h(u-theta)
h[0..100]=heav(u[j]-theta)
# define the special convolution operator
special k=conv(zero,101,5,w,h0)
u[0..100]'=-u[j]+k([j])
init u0=1,u1=1,u2=1
par theta=.1
done
```

NOTE that a common error is to write `k[j]` instead of `k([j])`. In *XPPAUT* , the brackets are ignored so that `k[10]` would be interpreted as `k10` which makes no sense since k is actually a function. However, `k([j])` is interpreted correctly since `[j]` is evaluated for `j=0,...100` as the integer `j`. The array that results from this convolution is zero offset so that the first element is `k(0)`.

Run this file and integrate it. Plot the array of values from `u0` to `u100` using the array plot to see the wavefront. (Either use the short cut by checking the boxes for `u0` and `u100` in the **Initial Data Window** and then clicking on the `Arry` button, or use the `Graphics Array` command and fill in the dialog box.) In the **Main Window** plot a few spatial profiles such as `u10,u30,u60` and measure the times of crossing $u = 0.5$ for `u30, u60` in order to measure the velocity of the wave.

We now turn to the convolution problem (6.4) We will define the weights as follows:

$$W(l) = \exp(-c|l|) - a\exp(-b|l|)$$

where c, a, b are positive parameters. Once again, here are the steps:

1. Define the table of weights using the variable `t` as the dummy.

   ```
   table w % 21 -10 10 exp(-c*abs(t))-a*exp(-b*abs(t))
   ```

2. Apply whatever function you want to the variables you want to convolve. (Optional)

3. Define the convolution with the `special` operator

   ```
   special k=conv(periodic,101,10,w,u0)
   ```

   (Here, I use periodic conditions on the array).

4. Define the right-hand sides using `k([j])`

   ```
    u[0..100]'=-u[j] + f(ce*k([j])-ci*delay(u[j],tau))
   ```

Here is the complete ODE file:

```
# nnetdelay.ode
# a neural net with delays
table w % 21 -10 10 exp(-c*abs(t))-a*exp(-b*abs(t))
par cp=4,cm=4,tau=0,r=3,ut=.25
par a=.5,b=.1,c=.25
init u[45..55]=.5
special k=conv(periodic,100,10,w,u0)
f(u)=1/(1+exp(-r*(u-ut)))
u[0..100]'=-u[j] + f(cp*k([j])-cm*delay(u[j],tau))
@ dt=.05,total=100,delay=10
done
```

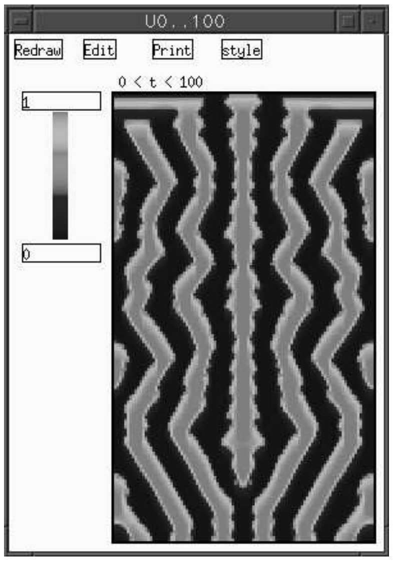


Figure 6.9. *Solutions to the delayed neural net equation with a delay of 1.*

As in all of our examples, there are 101 units. We will integrate this system for 100 time steps. We have initialized the middle 10 units to be 0.5. Because the network is periodic, we need to introduce a spatial inhomogeneity if we are to obtain any spatial patterns since homogeneous solutions are preserved. The last line tells *XPPAUT* that the maximum delay will be 10. By default $u_j(t) = 0$ for $t < 0$. We have initialized the delay parameter τ to be zero. Run this problem. Use the array plot to see the space-time behavior. Notice that the solution is a series of stripes; the medium has broken into a spatially periodic pattern. Now we will increase τ and see what happens. Set $\tau = .5$ and reintegrate the equations. The edges of the stripes have small damped oscillations. (Plot `u25` as a function of time to see this.) (**Question:** How do you know that these oscillations are real? - Make the timestep `DeltaT` smaller (`nUmerics  Delta T`) and note that they seem to damp out faster. You will find that eventually, they remain for small timesteps so that they appear to be true damped oscillations.)

Set $\tau = 0.75$ and solve again. It goes to a complex zig-zag pattern. Increase τ or change the parameter `cm` to see some other cool patterns. You may have to increase the amount of time to see the asymptotic state. Figure 6.9 shows a typical simulation.

Another animation. Let's create an animation of these patterns which treats the spatial array as a "rope" that evolves in time. If you want to understand animation files completely consult Chapter 8. For now type in the file `nnetdelay2.ani`:

```
# use with nnetdelay.ode
# a one liner!
```

```
line .05+.9*[0..99]/101;.05+.8*u[j];.05+.9*[j+1]/101;.05+.8*u[j+1];$BLACK;2
done
```

Click on `Viewaxes Toon` . The animation window pops up. Click on `File` in the animation window and load the above file, `nnetdelay2.ani`. Then press `Go` . Rerun the simulation with different parameters and look at the animation.

More spatial operators. There are other "spatial" operators which can be used in conjunction with tables to create fairly complex models. A chain of weakly coupled oscillators provides a good example of the `fconv` operator. Phase models arise from weak coupling and often have the form:

$$\frac{d\theta_i}{dt} = \omega_i + \sum_{k=-m}^{m} w(k)\Gamma(\theta_{i+k} - \theta_i) \tag{6.5}$$

where $\Gamma(\phi)$ is a $2\pi-$periodic function. (This function arises from the theory of averaging which we describe in detail in Chapter 9). In *XPPAUT* , the expression

$$R(j) = \sum_{l=-m}^{m} w(l) f(u(j+l), v(j)) \quad j = 0, \dots, n-1$$

is represented by the statement

```
special r = fconv(type,n,m,w,u,v,f).
```

This is similar to the `conv` operator but has additional arguments. The first 5 are exactly like in the `conv` operator. (`type` is either `zero,even` or `periodic`.) Additionally there is another array `v` and the name of a function of two variables, `f`. *XPPAUT* applies the function to $u(j+l)$ and $v(j)$ before multiplying by the weight $w(l)$ and adding up. Thus, to solve (6.5), we would define a weight, a function of two variables, and then the special convolution:

```
table w % 11 -5 5 exp(-abs(t))
gam(u,v)=ggamf(u-v)
special r=fconv(zero,40,5,w,theta0,theta0,gam)
```

Note that I use the same array for `u` and `v` in the fifth and sixth arguments of `fconv`; they need not be different. Thus

$$r(j) = \sum_{l=-5}^{5} e^{-|l|}\Gamma(\theta_{j+l} - \theta_j) \quad j = 0, \dots, 39.$$

Since I am interested in the relative phases that evolve, I will subtract off $d\theta_0/dt$ which is proportional to $r(0)$. Thus the ODE file for (6.5) is

```
# phschn.ode
# a chain of 40 phase oscillators
# in relative phase coordinates
```

```
table w % 11 -5 5 exp(-abs(t))
gam(u,v)=gamf(u-v)
gamf(u)=a0+a1*cos(u)+b1*sin(u)+b2*sin(2*u)
special r=fconv(zero,40,5,w,theta0,theta0,gam)
theta[0..39]'=r([j])-r(0)
par a0=.25,a1=.4,b1=1,b2=-.2
@ total=200,dt=.25,meth=cvode
@ nplot=5,yp=theta5,yp2=theta10,yp3=theta15,yp4=theta20,
@ xhi=200,ylo=0,yhi=10
done
```

NOTES: The first argument in `fconv` is `zero` since I want the chain to be a finite line of oscillators rather than a ring. Since the `fconv` function requires a function of two variables, I have defined `gam(u,v)=gamf(u-v)`. I subtract off `r(0)` from each of the equations so that the resulting solutions are the phases relative to `theta0.` I have used the stiff integrator, `CVODE` even though the problem is not stiff. I experimented with several different methods and this was the fastest. I have initialized the program so that there are four quantities plotted. Up to 9 can be so initialized.

Spatial profiles. Run this program and notice that all 4 plots seem to converge to a fixed value that is the relative phase. Thus, this particular equation system phaselocks. To see a spatial profile, get rid of all the plots by clicking `Graphics Remove all` (**G R**). Then transpose the data by clicking `File Transpose` and filling in the dialog box as:

`Column 1:` theta0
`NCols:` 40
`Colskip:` 1
`Row 1:` 750
`NRows:` 1
`RowSkip:` 1

and click on `Ok` . Now in the main window, click on `X vs t` and choose `theta0`. You will see that the oscillator in the middle is the most phase-advanced and the two edge oscillators fire last. Numerical simulations like this have suggested many theorems about coupled oscillator arrays. Change some of the parameters in the models. Can you break up the phase-locked solutions?

Non-convolution coupling. The examples presented above assume convolution coupling between the units. Suppose we wanted to simulate a population of 30 excitatory and 10 inhibitory cells. Let u_j, v_j be the activities of the respective excitatory and inhibitory cells. Then the equations are:

$$u_j' = -u_j + F_u(c_{ee} \sum_{k=0}^{29} w_{jk}^{ee} u_k - c_{ie} \sum_{k=0}^{9} w_{jk}^{ie} v_k) \quad j = 0, \ldots, 29$$

$$\tau v_j' = -v_j + F_v(c_{ei}\sum_{k=0}^{29} w_{jk}^{ei} u_k - c_{ii}\sum_{k=0}^{9} w_{jk}^{ii} v_k) \quad j = 0, \ldots, 9.$$

Note that here the matrices that multiply the variables are not square. *XPPAUT* has a way of dealing with this type of interaction. Once again, the weights are defined by lookup tables. Since w^{ee} is 30×30 the lookup table will be 900 elements. The first 30 are row 1, the next are row 2, etc. The matrix $w^{ie}$is 30×10 so it requires a table of length 300; the first 10 elements are row 1, the next row two, and so one. For the present example, we will assume that w_{jk} are uniformly distributed on $[0, r]$ such that the mean sum of any row is 1. Given the table, w that corresponds to a matrix $n \times m$ and an array of variables of length m starting at `z0`, then the declaration:

```
special q=mmult(m,n,w,z0)
```

produces a function `q` such that

```
q(j) = sum(0,m-1)of(w(jm+i')*shift(z0,i'))
```

Thus, the ODE file for this problem is

```
# randnet.ode
table wee % 900 0 899 ran(1)/15
table wie % 300 0 299 ran(1)/5
table wei % 300 0 299 ran(1)/15
table wii % 100 0 99 ran(1)/5
special see=mmult(30,30,wee,u0)
special sie=mmult(10,30,wie,v0)
special sei=mmult(30,10,wei,u0)
special sii=mmult(10,10,wii,v0)
fu(x)=1/(1+exp(-(x-uth)))
fv(x)=1/(1+exp(-(x-vth)))
u[0..29]'=-u[j]+fu(cee*see([j])-cie*sie([j]))
v[0..9]'=(-v[j]+fv(cei*sei([j])-cii*sii([j])))/tau
par cee=10,cie=8,uth=1.55,cei=13,cii=8,vth=2,tau=4
aux uu=sum(0,29)of(shift(u0,i'))/30
aux vv=sum(0,9)of(shift(v0,i'))/10
@ autoeval=0,total=50
done
```

I have added the line `autoeval=0` which tells *XPPAUT* not to re-evaluate the tables every time the parameters are changed. Thus, the weights are **fixed** as we vary the other parameters. The default for *XPPAUT* is to re-evaluate tables whenever parameters are changed. Since the tables here are random, we don't want to alter them. I have also added the population averages of the excitatory and inhibitory populations, `uu,vv` as auxiliary variables. Curiously enough, this network seems to synchronize and produces rhythmic output quite robustly. As an extra credit problem, simulate the mean field model for this equation and compare it to the full model. The mean field equations are:

$$u' = -u + f_u(c_{ee}u - c_{ie}v), \qquad \tau v' = -v + f_v(c_{ei}u - c_{ii}v).$$

Some tricks using the network functions

There are other network functions which are more flexible than the convolutions. The `sparse` operator allows you to create networks of length n in which each element j is connected to m other elements i in the network with weight w_{ji}. This operator requires that you provide 2 tables of length nm which give the connectivity matrix and the strength of the connections. As with the above example, two-dimensional arrays are stored in row by row. (That is, the rows of the matrix are appended in order to form a one-dimensional array.) For example instead of (6.4) consider the random network:

$$u_j' = -u_j + f(c^+ \sum_{i \in C_j} w_{ji} u_i - c^- u_j(t-\tau)) \quad j = 0, \ldots, 99$$

where C_j is a randomly chosen set of 10 indices and w_{ji} is a random number between 0 and 1. In order to make the *XPPAUT* file for this, we will need to define two tables. One of them is the connectivity table which gives the indices of the cells that cell j is connected to. The other table contains the weights. The connection matrix is $10 \times 100 = 1000$ entries whose elements are the indices 0-99 since there are 100 units labeled `u0` through `u99`. The following does the trick:

```
table con % 1000 0 999 flr(ran(1)*100)
```

The function `flr(x)` returns the greatest integer less than or equal to `x`. The actual weights are randomly distributed between 0 and 1. The special function `k=sparse(n,m,w,c,u0)` returns `n` values:

$$k(j) = \sum_{i=0}^{m-1} w(j \cdot m + i) \mathrm{shift}(u0, c(j \cdot m + i))$$

where c is the connectivity array. Thus the ODE file is written:

```
# sparse.ode
# a random sparse network with delays
# each is connected to 10 others
table con % 1000 0 999 flr(ran(1)*100)
table w % 1000 0 999 ran(1)
par cp=1,cm=6,tau=3,r=3,ut=.25
special k=sparse(100,10,w,con,u0)
f(u)=1/(1+exp(-r*(u-ut)))
u[0..99]'=-u[j] + f(cp*k([j])-cm*delay(u[j],tau))
@ dt=.05,total=100,delay=10
@ autoeval=0
done
```

It is similar to the file for a regular network with the exception of the definitions of the the weights and the connectivity. Note, again, we have added the statement `autoeval=0` in the penultimate line to tell *XPPAUT* that the tables should **not** be recomputed automatically each time a parameter is chosen.

Before running this file, there is one more point. In many neural network models, self-connections are not allowed. So, how could we modify the connectivity function to avoid this? Given an index i we can add to this a randomly chosen number from 1 to 99 and mod out by 100. Thus, i would never be chosen. The following fragment of code would replace line 4 in the above ODE file:

```
table con % 1000 0 999 mod(flr(t/10)+1+flr(99*ran(1)),100)
```

The `flr(t/10)` is just the index i since the first 10 entries are connections to 0, the next 10 to 1, and so on. The term `1 + flr(99*ran(1))` generates a random integer from 1 to 99.

Run this file and look at the space-time evolution. It appears that there is no pattern formation other than the small randomness. Essentially, the whole thing appears like a mean field model. This suggests a theorem which is that the sparse excitatorily coupled oscillating network leads to synchronous oscillations as N, m get large. Compare the solutions to this to the mean field equation:

$$u' = -u + f(5c^+u - c^-u(t-\tau))$$

where the factor of 5 comes from the fact that each cell receives 10 inputs with mean value 0.5.

A dendritic tree. The bulk of most neurons consists of the dendritic tree of the neuron. One approach to simulating the tree of the neuron is to break it into a series of small connected cylinders. Each cylinder has a specified length and diameter as well as a specified axial resistance and transmembrane resistance. The former depends on the diameter and length of the cylinder while the latter depends on the surface area excluding the faces. Ohm's law then allows one to approximate the dynamics of the dendritic tree by an electrical circuit which in turn leads to a series of differential equations for the voltage of each cylinder. Mark Fenner, a student in one of my classes, has written some C++ code which allows one to take a list of endpoints of the cylinders and their diameters and create an *XPPAUT* file that simulates the dendritic tree. The C++ program produces four files: two tables that describe the connectivity and the weights, an animation file which draws the tree, and the ODE file to drive the simulation. The key utility of the program is the production of the connectivity and weight matrices as well as the animation file. The ODE file produced is simple and assumes passive dendrites. It is trivial to alter this by hand to make some of the compartments active. Each cylinder can connect to no more than 4 other cylinders since branching always consists of splitting into two. The connectivity is of the form:

$$\sum_i g_{ji}(V_i - V_j) = \sum_i g_{ji}V_i - \left(\sum_i g_{ji}\right) V_j.$$

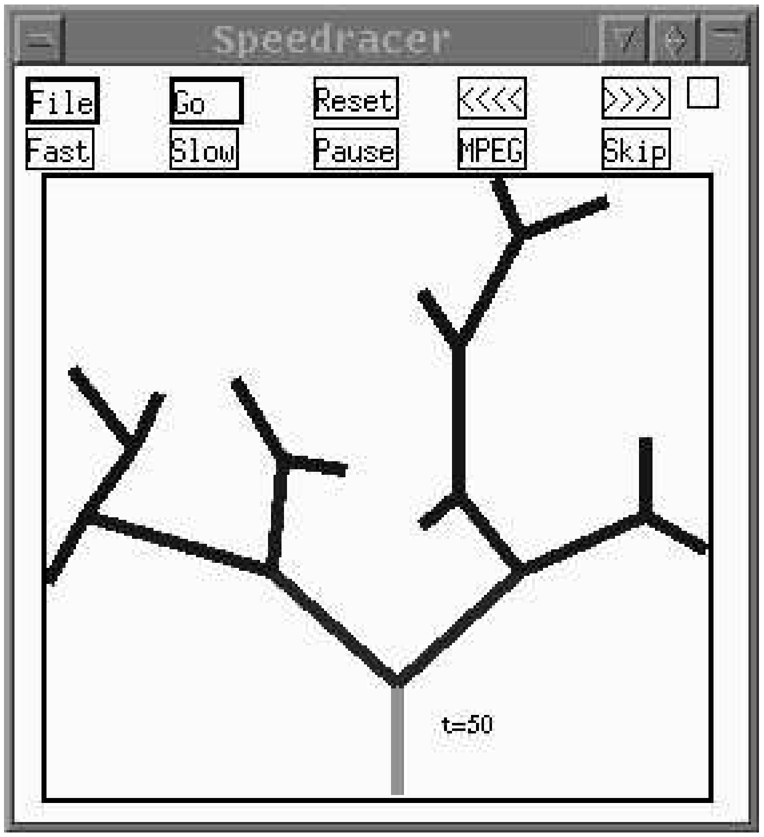


Figure 6.10. *A dendritic tree rendered in the animation window*

Thus for N compartments, two tables of length $5N$ are created that describe the weights and the connectivity. We pad the weights for compartments that connect to fewer than 4 others with zeros. The input file for this nice little utility consists of the (x, y) coordinates for the endpoints of each cylinder and the diameter. The program uses the endpoint information to decide which cylinder is connected to which and it uses the dimensions of the cylinder to determine the relative weights of the coupling. The lengths are scaled and then an animation file is created that draws the cylinders. The animation files draws each cylinder color coded according to some user-specified function. The C++ code and some examples are available on the *XPPAUT* web site. An example is shown in figure 6.10.

One-dimensional cellular automata. Cellular automata enjoyed some popularity a number of years ago and are still quite amusing to play with. A simple class of model CA consists of a line of cells that take on values of 0 or 1. At each time step the sum of m neighbors is taken (which is thus between 0 and $2m + 1$) and a rule is provided that maps each possible sum to either 0 or 1. Typically, the boundary conditions are periodic. Fix m and let f be a map from $\{0, 1, \ldots 2m+1\}$ to $\{0, 1\}$. Then, we can view a one-dimensional cellular automata as the discrete dynamical system:

$$u^{t+1}(j) = f\left(\sum_{i=-m}^{m} u^t(j-i)\right).$$

Since the domain of f is generally a small number of discrete points, we will read in f as a lookup table. We will define a weight matrix of all 1's which we convolve with u from $-m$ to m and then apply f to the resulting sum. Here, then, is an *XPPAUT* file for a CA of 101 cells which looks at 2 neighbors to the left and right as well as the value of the cell itself. The rules are in the form of a table which the user can modify externally and read in.

```
# ca100.ode
# 101 cell CA model
# here is the function for the new value
table f ca5.tab
# add up with equal weight 5 cells
table wgt % 5 -2 2 1
# The convolution is created
special k=conv(periodic,100,2,wgt,u0)
# here is the update
u[0..100]'=f(k([j]))
# make sure XPP knows it is discrete time
@ total=200,meth=discrete
done
```

NOTES: The table that defines f the rule is read in as a file. Here is the table file:

```
6
0
5
0
0
1
0
1
1
```

The first line tells *XPPAUT* how many points. Since the sum of neighbors can be $0, 1, \ldots, 5$ there are 6 possible sums. The next two lines tell *XPPAUT* the domain of the function. The last 6 lines are the rule itself. Thus for example, if the sum of neighbors is 3, then the value is $f(3) = 0$ while if the sum is 5, the returned value is $f(5) = 1$.

Fire up *XPPAUT* with this file. Click on `Initialconds Formula` (**I U**) and choose `U[0..100]` as the first and `heav(ran(1)-.5)` as the formula. This turns on half the cells. Hit `Enter` at the next prompt and the simulation will be run.

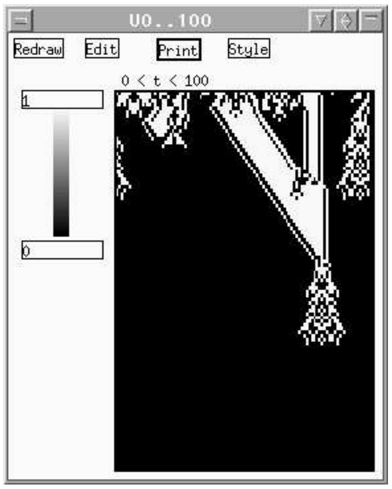


Figure 6.11. *Cellular automata simulation*

Use the array plot to look at the space time evolution of the automata. (Here is the easy way: in the **Initial Data Window** click on the square next to `u0`. Scroll down to `U100` and click in its square. Then click on the `arry` button at the bottom of the **Initial Data Window** .)

Using a text editor, create a new table file called `ca5_III.tab` that has the form:

```
6
0
5
0
0
1
1
1
0
```

This rule produces a so-called class III cellular automata and is quite pretty. In *XPPAUT* click on `nUmerics lookUp` (**U K**) and type in `f` for the lookup table name. For the filename, type `ca5_III.tab` and click `Enter` twice and then `Esc` to get to the main menu. You have just replaced the original lookup table with the new one you edited. Click **I G** to rerun the simulation. Click on `Redraw` in the array window to redraw the result. You should see a series of triangles.

Play around with your own tables and change the initial conditions. See if you can classify all the possible types of patterns. Note that there are only 64 possible rules. Maybe you can get a MacArthur award and invent MATHEMATICA just like the guy who did this research!

6.2.4 Exercises

1. Try initial data such as $u(x,0) = \cos(n\pi x)$ for higher values of n and verify that the steady state is reached much more quickly for higher values of n in the cable equation, Eq (6.2).

2. The steady state for 6.2 is

$$\begin{aligned} 0 &= D\frac{d^2u}{dx^2} - u \\ c_0 &= a_0 u(0,t) + b_0 u_x(0,t) \\ c_L &= a_L u(L,t) + b_L u_x(L,t). \end{aligned}$$

which is just a second order equation with boundary conditions. Use *XPPAUT* to solve this boundary value problem for a variety of boundary conditions including the one we used above. Figure out what values to use for L and D in order to quantitatively compare it to the steady states of the PDE. If you get stuck, here is the ODE file but try to create it before looking:

```
# cable boundary value problem
# 0 = D u'' - u
# c0 = a0 u(0) + b0 u'(0)
# cl = al u(l) + bl u'(l)
u'=up
up'=u/D
par D=1,c0=1,a0=1,b0=0,cl=0,al=0,bl=1
bndry a0*u+b0*up-c0
bndry al*u'+bl*up'-cl
@ total=10.001,dt=.01
@ xhi=10.001,yhi=1.00,ylo=0
# click on Bndryvalue Show to solve this!
# try different values for the parameter
done
```

3. Solve the PDE:

$$\frac{\partial u}{\partial t} = D\frac{\partial^2 u}{\partial x^2} + u(1-u)$$

with $D = 1$ on a domain of length 20 with 100 grid points, a time step of `dt=0.025` with Neumann boundary conditions, $u_x(0,t) = u_x(L,t) = 0$ for $0 < t < 20$. Initialize the first grid point to 1 and all the rest to 0. (Hint: copy and modify the cable PDE.) Now try the following interesting experiment. Use the `Initialconds formula` command to initialize `u1..u100` to `exp(-ct)` where `c=1, 0.1, 0.05`. By plotting `u80` and `u90` in the same window, show that the velocity of this front depends on the initial conditions. (Measure the time difference between crossing of $u = 0.5$.) Thus show that the front is slower with the sharper initial data.

4. Repeat the previous exercise with $u(1-u)$ replaced by $u(u-a)(1-u)$ where $a = .02$ and see that the velocity is independent of the initial condition; it is unique.

5. Obtain a traveling wave solution to the Morris-Lecar equations by solving the initial value problem in the excitable regime. You should try using the equations and parameters used in the shooting exercise. Use Neumann boundary conditions and work on a domain of size 10 with 100 grid points. Assume that the diffusion coefficient is 0.1. Initialize the voltages to rest, $V = -0.4$ and initialize the first four points in the grid to 0 to initialize the wave. Run the simulation and observe the traveling wave. Try computing the velocity by finding the amount of time it takes to travel m grid points. The velocity is $mh/(\sqrt(d)T)$ where $h = 10/100$ is the grid size and T is the time to travel the m grid points. Compare this to the velocity you obtained by shooting. As in the previous exercises, try to make the ODE file yourself before looking at the answer below:

```
# morris-lecar PDE
par d=.1,h=.1
# note that I have to change the names of the parameters
# so that I can use v1,v2, etc as the voltage grid points
#
params va1=-.01,va2=0.15,va3=0.1,va4=0.145,gca=1.33,phi=.333
params vk=-.7,vl=-.5,iapp=.04,gk=2.0,gl=.5,om=1
minf(v)=.5*(1+tanh((v-va1)/va2))
ninf(v)=.5*(1+tanh((v-va3)/va4))
lamn(v)= phi*cosh((v-va3)/(2*va4))
f(v,w)=gl*(vl-v)+gk*w*(vk-v)+gca*minf(v)*(1-v)+iapp
g(v,w)=lamn(v)*(ninf(v)-w)
v0=v1
v[1..100]'=f(v[j],w[j])+d*(v[j+1]-2*v[j]+v[j-1])/(h*h)
v101=v100
w[1..100]'=g(v[j],w[j])
init v1=0,v2=0,v3=0,v4=0
init v[5..100]=-.4
@ total=50
done
```

6. In this exercise, we will use a classic reaction-diffusion model to illustrate pattern formation. Turing (1955) proposed that spatial instabilities could arise if coupled chemical systems had differing diffusion constants. The equations for a general two-component reaction-diffusion equation are:

$$\frac{\partial u}{\partial t} = f(u,v) + D_u \frac{\partial^2 u}{\partial x^2}$$
$$\frac{\partial v}{\partial t} = g(u,v) + D_v \frac{\partial^2 v}{\partial x^2}$$

with boundary conditions such as Neumann or periodic or Dirichlet. The Brusselator is an example of such a system where (f, g) are

$$f(u, v) = a - (b+1)u + u^2 v, \quad g(u, v) = bu - u^2 v.$$

The spatially homogeneous equilibrium is $(u, v) = (a, b/a)$ and it is stable to spatially homogeneous perturbations as long as $b < 1 + a^2$. However, if this condition holds and D_v/D_u is sufficiently large, then the spatially homogeneous solution is unstable to nonhomogeneous perturbations. New solutions arise which are spatially inhomogeneous. Here is an ODE file for the Brusselator:

```
# the brusselator reaction-diffusion model
# brusspde.ode
# reaction terms
f(u,v)=a-(b+1)*u+v*u^2
g(u,v)=b*u-v*u^2
par a=1,b=1.5
#
# equations
u0=u1
u[1..100]'=f(u[j],v[j]) + du*(u[j+1]-2*u[j]+u[j-1])/(h*h)
u101=u100
#
v0=v1
v[1..100]'=g(u[j],v[j]) + dv*(v[j+1]-2*v[j]+v[j-1])/(h*h)
v101=v100
# spatial parameters
par  h=.2,du=.1,dv=4
# initial data
init u[4..100]=1
init u[1..3]=1.2
init v[1..100]=1.5
# numerical stuff
@ meth=cvode,atol=1e-5,tol=1e-6,total=400,dt=.25
done
```

It is very similar to the simple cable model discussed above. Because diffusion equations tend to be numerically stiff (due to the division by h^2), I have added a line at the end of the file to use the stiff integrator, `CVODE` to specified tolerances with a timestep of 0.25 for a total of 400 time units. Run this and look at the steady states. Change D_v, making it smaller and verify that the spatially homogeneous state becomes stable again. Note that you can make this much faster by using the banded method and rewriting the discretized system like the example (6.3).

7. As a last PDE example, consider the Grey-Scott equations. This is an example

which makes it very clear why you want to use the banded option. Here is the file `gs.ode`

```
# grey scott equations
par d=.25,al=.1,a=2,h2=100
u1'=(d*(u2-u1)+a*u1^2*z1-u1)/al
z1'=h2*(z2-z1)-u1*u1*z1+1-z1
%[2..199]
u[j]'=(d*(u[j+1]-2*u[j]+u[j-1])+a*u[j]*u[j]*z[j]-u[j])/al
z[j]'=h2*(z[j+1]-2*z[j]+z[j-1])-z[j]*(1+u[j]*u[j])+1
%
u200'=(d*(u199-u200)+a*u200^2*z200-u200)/al
z200'=h2*(z199-z200)-z200*(1+u200*u200)+1
init z[1..200]=1
init u1=1.2
@ meth=cvode,tol=1e-5,atol=1e-4,bandup=2,bandlo=2
@ dt=.25,total=200,xhi=200,yhi=25,yp=u100
done
```

Run this and time how long it takes. Then click on `nUmerics Method` and choose `Cvode` but turn off the banded option. Rerun the simulation – make sure you have a copy of War and Peace to read in the meantime. (It is about 30 times slower.)

8. Explore the behavior of the continuous approximation to the class III cellular automata:

$$u_j' = -u_j + F(\frac{1}{11}\sum_{k=-5}^{k=5} u_{j+k}(t-\tau))$$

with periodic boundary conditions and

$$F(u) = \frac{1}{(1+\exp(-a(u-z_1)))(1+\exp(a(u-z_2)))}$$

with $0 < z_1 < z_2 < 1$ and $a > 0$. Note that you will have to define the delayed variables as fixed variables. Try $\tau = 4, a = 20, z_1 = .2, z_2 = .6$ and initialize a few variables to 1. Integrate it for a while — to speed things up, use Euler. Here is an ODE file

```
# ctsca.ode
# continuous approximation to class 3 CA
du[0..100]=delay(u[j],tau)
table w % 11 -5 5 1/11
f(u)=1/((1+exp(-a*(u-z1)))*(1+exp(a*(u-z2))))
par tau=4,a=20,z1=.2,z2=.6
init u[45..55]=1
special k=conv(period,101,5,w,du0)
```

```
u[0..100]'=-u[j]+f(k([j]))
@ total=100,delay=10,meth=euler
done
```

9. Write an ODE file for Nick Swindale's model for occular dominance columns in one-dimension:

$$\frac{\partial n(x,t)}{\partial t} = n(x,t)(1-n(x,t))(\mu \int_D W(x-y)n(y,t)\ dy - \nu).$$

Choose 101 points, $\mu = 0.5$, $\nu = 1.3$,

$$W(x) = e^{-0.1x^2} - 0.35e^{-.025x^2}.$$

Approximate the continuous convolution by a sum from -15 to 15 on a periodic domain. Use random initial data between 0 and 1. Here is part of the ODE file

```
table wa % 31 -15 15 exp(-a*t*t)-c*exp(-b*t*t)
special k=conv(periodic,101,15,wa,n0)
```

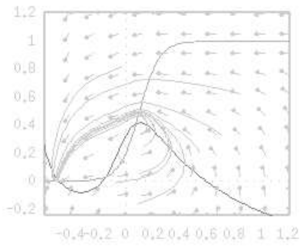
10. Solve the globally coupled system:

$$u'_j = -u_j + d(u_{j-1} - 2u_j + u_{j+1}) + ku_j/(1+\bar{u}^2)$$

where

$$\bar{u} = \frac{1}{N+1}\sum_{j=0}^{N} u_j.$$

Take $N = 50, d = 1$, and vary k between 1 and 4. Use random initial data and reflecting boundary conditions. Does this suggest any kind of spontaneous pattern formation? Can you analytically determine the stability of the origin for this?



Chapter 7

Using AUTO – bifurcation and continuation

For many people, the main reason to use *XPPAUT* is because it provides a fairly simple interface to the continuation package, AUTO (Doedel,1986). AUTO now has a GUI interface in the distributed version, but still requires that you write FORTRAN code to drive it. *XPPAUT* allows you to use most of the common features of AUTO including following fixed points, periodic orbits, boundary value problems, homoclinic orbits, and two parameter continuations. This chapter will show you how to solve a variety of problems with AUTO. Keep in mind that AUTO is a tricky program and can fail in a dramatic fashion. Nevertheless, it is a powerful tool and even though it is 15 years old, it is still better than most other similar programs when it comes to tracking periodic solutions and solving BVPs.

Many physical and biological systems include free parameters. One of the goals of applied mathematics is to understand how the behavior of these systems varies as the parameters change. This is a numerically challenging task and there is generally no way to systematically explore a system as a function of all it's parameters. One of the first things a modeler should do is to reduce the number of parameters by either fixing those which are best known (e.g, gravity) and by making the problem dimensionless. Dimensional analysis is very useful and the reader should consult any number of books which describe this technique (Edelstein-Keshet 1989, Murray 1992). Assuming that you can reduce the free parameters to a manageable number, there are some useful tools for exploring how a dynamical system changes as these parameters vary. The most straightforward technique is called *continuation* in which a particular solution (such as a fixed point or limit cycle) is followed as the parameter changes. AUTO, (which is described in this chapter) provides some very powerful algorithms for the continuation of fixed points and periodic solutions to differential equations. The stability of the particular *branch* of solutions is readily obtained by analyzing the linearization. This, too is automatically accomplished by AUTO. If a fixed point or limit cycle exhibits a change in its stability, this is often a sign that new qualitatively different behavior could happen. For example, new fixed points might emerge from the branch of solutions or a limit cycle may emerge. These qualitative changes in the local and global behavior are called *bifurcations* and their detection during continuation is the subject of much mathematical research. AUTO provides a number of tools for the automatic detection of bifurcations of fixed points and limit cycles. A thorough description of local and global bifurcations can be found in the excellent book by Kuznetsov.

Consider a differential equation of the form:

$$\frac{dx}{dt} = F(x, \mu), \qquad x \in R^n \tag{7.1}$$

where μ is a parameter. Suppose that (7.1) has a fixed point, (x_0, μ_0). Let $A(\mu_0) = A_0$ be the linearized matrix for F with respect to x about the fixed point (x_0, μ_0). There are two important insights gained by looking at A_0. First, if A_0 is invertible, then we can follow the fixed point x_0 as a function of the parameter μ in some neighborhood of μ_0 and this is the *only* fixed point near x_0. This fact is a consequence of the implicit function theorem. The second point is that if none of the eigenvalues of A_0 lie on the imaginary axis, then the behavior of (7.1) near the fixed point x_0 is the same as the behavior near the origin of the linear differential equation:

$$\frac{dy}{dt} = A_0 y. \tag{7.2}$$

In particular, the stability of the two systems is the same. This is why we can determine stability of fixed points by linearizing. However, if one or more eigenvalues of A_0 have a zero real part, then the local behavior of the nonlinear system is not necessarily the same as the linear system. Convince yourself of this fact by looking at the following three systems which all have the same linearization:

$$\begin{aligned} x' &= -y - y^3 \qquad & y' &= x \\ x' &= -y - x^3 \qquad & y' &= x \\ x' &= -y + x^3 \qquad & y' &= x \end{aligned}$$

Thus, if we vary μ and one or more eigenvalues of $A(\mu)$ crosses the imaginary axis, we expect that there will be a *qualitative* change in the behavior of (7.1). The analysis of this change in behavior near the fixed point is the goal of *local bifurcation theory.*

Similar points can be made for discrete dynamical systems:

$$x_{n+1} = F(x_n, \mu), \tag{7.3}$$

but the critical set for the eigenvalues of $A(\mu)$ is the unit circle rather than the imaginary axis. Consider a limit cycle solution to (7.1) and take a local Poincare section. Then, this defines a map and the limit cycle is a fixed point of the map. Thus, the local bifurcations of limit cycles are found by studying the local bifurcations of the associated Poincare map.

One of the beauties of local bifurcation theory is that the behavior of the systems, (7.1) and (7.3) is the same as certain low-dimensional polynomial systems called *normal forms.* We briefly mention the main bifurcations of interest:

- Bifurcations at a zero eigenvalue for differential equations. When A_0 has a zero eigenvalue, then new fixed points often arise. These can be transcritical, pitchfork, or saddle-node bifurcations. Only the latter is generic; the other two occur when there is symmetry. The transcritical and pitchfork are *branch point* bifurcations and AUTO will automatically detect and follow them. The saddle-node or fold or limit-point bifurcation occurs when a branch of solutions bends around.
- The Hopf bifurcation. This occurs when A_0 has a pair of imaginary eigenvalues. Generically, a small amplitude limit cycle will emerge from the branch of fixed points.
- Period doubling bifurcation. This occurs for maps when an eigenvalue of A_0 is -1. For a limit cycle, the result is that the period of the limit cycle doubles.
- Torus bifurcation. When the eigenvalues of A_0 for a map cross the unit circle at values other than ± 1, $\pm i$ or the cube roots of 1, then a torus bifurcation arises. Chaos and other complex behavior often appear.

AUTO detects all of these bifurcations and in some cases can create a two-parameter curve along which the bifurcation occurs.

In the next few sections, we show how to use *XPPAUT* and AUTO together on a variety of differential equations, maps, and boundary value problems.

7.1 Standard examples.

As a first example, we consider the normal form for a cusp bifurcation:

$$x' = a + bx - x^3.$$

Let's pick $a = b = 1$ and let *XPPAUT* find the root by integrating the equation. Make an ODE file for this problem and run it. Integrate it and then integrate it again using the `Initialconds Last` (**I L**) command to pick up where you left off. You should have settled onto a fixed point of about 1.3247. Click on `File Auto` to bring up the **AUTO window**. (See figure 7.1.) In this window, there are additional menu items and several windows. The lower left shows a circle. The number of crosses inside (outside) the circle corresponds to the number of stable (unstable) eigenvalues for a solution. The two windows at the bottom are respectively information about the computed points and the hints or tips window. This bottom window also tells you the coordinates of the main graphics window. The main window is where the relevant bifurcation diagrams are drawn.

Before you can use AUTO, you must prepare your system for it. You must start your bifurcation analysis from either a fixed point of your model, a periodic orbit, or a solution to a boundary value problem. We have already done this for the present example.

Once you have brought up the **AUTO window** , you should do the following steps:

- Use the `Parameter` item to tell AUTO the list of 5 or fewer parameters that

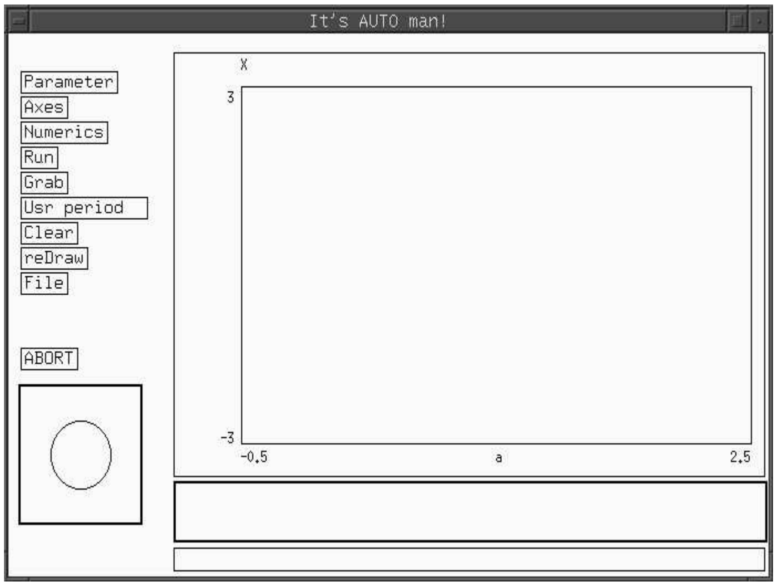


Figure 7.1. *The AUTO window*

you will vary for the session. Choose the names of any of your parameters; the default is the first 5 that you have defined in the ODE file.

- Use the `Axes` command to tell *XPPAUT* what parameters are varied and what is to be plotted as well as the range of the graphs.
- Use the `Numerics` command to define all the AUTO numerical parameters such as the direction to go, output options, step size, etc.
- Use the `Run` command to run the bifurcation.

Once you have run a bifurcation, use the `Grab` command and the arrow keys to move around the diagram and the `File` command to save things or reset AUTO. The commands `Clear` and `reDraw` just erase and redraw the screen respectively. The `Usr period` item allows you to tell AUTO to save certain points such as oscillations of a certain period or points when a parameter takes a particular value.

AUTO is essentially independent of *XPPAUT* but there is some communication back and forth. For example if you grab a point in the **AUTO window** , then the state variables are loaded as initial data into *XPPAUT* and the parameters are changed as well. Similarly, when you start fresh in AUTO, the initial fixed point or orbit is computed in *XPPAUT* . Finally, you can import a bifurcation diagram from AUTO into *XPPAUT* and plot this. We will use this feature later to show bursting in a biological model.

With these preliminaries, let's compute the bifurcation diagram for our cusp model. The main parameter is a and the secondary parameter is b. Since there are only two parameters, there is no need to invoke the `Parameter` item in the **AUTO window** . We eventually want to compute a two-parameter bifurcation diagram; to do this, we must first compute a one-parameter diagram. From here on in, all commands that I mention concern the items in the **AUTO window** . Click on `Axes` and choose `HiLo` ; this option plots both the maximum and minimum of orbits which for the present problem is irrelevant since there are no periodic orbits. In the resulting dialog box, you tell AUTO the variable to plot, the main parameter (for 1-parameter diagrams) and the second parameter (for 2-parameter diagrams) as well as the dimensions of the plot. Fill it in as follows:

Y-axis: X
Main Parm:: a
2nd Parm:: b
Xmin:: -2.5
Ymin:: -3
Xmax:: 2.5
Ymax:: 3

and then click `Ok` . We have told AUTO that the main bifurcation parameter is `a` and that `X` is the variable to plot. The graph will show `a` on the horizontal axis and `x` on the vertical with a range of $[-2.5, 2.5] \times [-3, 3]$. Next click on the `Numerics` item. Change only one item in this dialog box, `Par Min` , the minimum value of the parameter, which you should change from 0 to -2. `Par Min` and `Par Max` tell AUTO how far to continue solutions. Then click `OK` . Now you are ready to run. Click on `Run Steady state` . (This tells AUTO you are continuing along a steady state or fixed point of the system.) A thick black line will go across your screen toward the right. If nothing happens, you probably forgot to set a good initial guess for the starting point. You should make sure that you are on a fixed point in *XPPAUT* ; for $a = 1, b = 1$ the fixed point is about 1.3247. Assuming that you have a curve, you can see that it is pretty boring and that the curve was produced for a increasing. Click on `Grab` to move around on the bifurcation diagram. You should see a cross on the screen. (If you don't see a cross on the screen, then, next time you run *XPPAUT* , run it with the `-xorfix` option which should fix the lack of a cross. By default it runs without requiring this option on Linux and Windows.) Use the left and right arrow keys to move the cross around. The stability circle shows the eigenvalue status. The fact that the drawn curve is solid implies the fixed point is stable; unstable fixed points are drawn with thin lines. AUTO specially marks certain points with small crosses and numbers. You can jump to these by tapping **Tab** . As you move around the diagram, the text area beneath the diagram gives you a summary of information about the current location such as the value of the parameter, the state variable, the period and the point type for special points. There are many pointtypes in AUTO. In this example, you will see only `EP` which means endpoint. The one you never want to see is `MX` since this means AUTO has had trouble and maxed out on some sort of numerical iteration. Sometimes this can be rectified by changing the numerical parameters. You can exit from `Grab` mode by tapping `Esc` which does **not** grab the point or by tapping `Enter` which loads the point into memory. *AUTO can generally be restarted only from special points.* So, lets make this diagram extend to the left. Grab the first point in the diagram – `Grab` `Enter` . Click on `Numerics` and change `Ds` from `0.02` to `-0.02` and click on `OK` . This tells AUTO you want to change the direction since the sign of the numerical parameter `Ds` determines the starting direction for AUTO. Now, click on `Run` . You should now see a sideways cubic curve. In particular, there are two points labeled 3 and 4 occurring at $a = \pm 3.84$. If you click on `Grab` , you will discover that the two special points are labeled `LP` which means that they are limit points, also known as fold points. Note that as you move past them, the eigenvalue goes through the unit circle. The curve of fixed points along the middle branch is unstable.

Grab the leftmost limit point, #3. We will now track this fold point as a function of the second parameter, b, producing a two-parameter bifurcation curve. Click on `Axis` and select `Two param` . Change `Ymin` from -3 to -0.5. Since the plot type is `Two Par` the vertical axis contains the secondary parameter (`b`) and the horizontal, the main parameter (`a`). Click on `OK` . Click on `Run` . You should see a thin curve that rises and goes to the left. This is a curve of fold points; along this curve in (a, b) space, the original system has a fold.

Once again, invoke the `Grab` item and click on `Tab` until you have selected the limit point #3 again. (Look at the bottom of the **AUTO window** to see which points you move through.) We now want to change the direction of bifurcation to get the rest of the curve. Click on `Numerics` and change `Ds` from -0.02 to 0.02, thus forcing AUTO to make the main parameter increase. Click on `Run` and you will get the rest of the curve; it is a nice cusp. For each (a, b) inside the cusp, there are three fixed points and for (a, b) outside the cusp, there is only one fixed point.

Click on `Axes` `Last 1Par` and then on `reDraw` . You will get the one-parameter diagram again. Click on `Axes` `Last 2 par` `reDraw` and you will see the cusp. This is a shortcut to go between the most recently defined one- and two-parameter views. Click on `File` `Postscript` and you can make a postscript version of the diagram. Figure 7.2 shows the one-parameter and two-parameter diagrams.

7.1.1 A limit cycle

We will start on a limit cycle in this example. The limit cycle is essentially what is called an *isola* since it is not connected to a main branch of fixed points. Here are the equations:

$$x' = xf(R) - y \quad y' = yf(R) + x,$$

where $f(R) = 0.25 - a^2 - (R-1)^2$ and $R = x^2 + y^2$. There is one parameter in this problem, a and for $-.5 < a < .5$, a pair of limit cycles, one stable and one unstable. This can be seen by rewriting the system in polar coordinates in which case it becomes

$$r' = rf(r^2) \quad \theta' = 1.$$

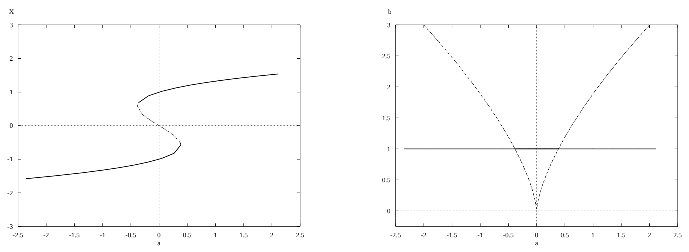


Figure 7.2. *One and two parameter bifurcation diagrams for the cusp ODE, $x' = a + bx - x^3$. The one-parameter diagram has $b = 1$. The two-parameter diagram also shows the artifactual line at $b = 1$ from the one-parameter continuation.*

Thus, when $f(A) = 0$, there is a limit cycle with magnitude $\sqrt{A}$. Starting with the parameter value $a = 0$, we see that $A = 1.5$ so that,

$$x(t) = \sqrt{1.5}\cos t \quad y(t) = \sqrt{1.5}\sin t$$

is a solution. (I will leave the verification of this as an exercise to the reader as well as the stability of the cycle. The reader should also show that for $a = 0$, another solution is $A = 0.5$ leading to an unstable limit cycle. Finally, the reader should prove that $(x, y) = 0$ is an asymptotically stable fixed point for all a.)

Here is how to start AUTO from a limit cycle. Note first, that this does not always work, AUTO has much more trouble starting from a periodic solution than from a fixed point.

1. Integrate the equations until a nice limit cycle is formed.

2. Determine the period either by looking at the data browser or by using the mouse to measure peak-to-peak distance in the oscillation. (Hints: In the **Data Viewer** , click on the button `Find` , choose a variable, and then pick a real big number. *XPPAUT* will then find the value of the variable closest to this big number, in other words, the maximum. Then click on `Get` which loads this point as an initial condition. Integrate again and look for the time of the next maximum. This is the period. Alternatively, hold the mouse in the window with the button pressed down and move it around – you can read off the coordinates at the bottom of the window.)

3. Change the total amount of integration time to the approximate period (`nUmerics Total`) and integrate the equations one more time.

4. Fire up AUTO and set up the bounds for the parameter as well as the graphical viewpoint.

5. In the **AUTO window** , click on `Run Periodic` .

6. Keep your fingers crossed.

Let's apply these techniques to the isola problem. Here is the ODE file:

```
# an isola of limit cycles
# isola.ode
f(r) = .25-(r-1)^2-a^2
r=x^2+y^2
x'=f(r)*x-y
y'=f(r)*y+x
par a=0
init x=1.224,y=0
@ ylo=-1.5,yhi=1.5
done
```

Run *XPPAUT* and integrate the equations. Use the mouse to see that the period is roughly 6.3 (it is actually 2π). Change the integration time to 6.3 and run again by clicking `Initialconds Last` starting at the end of the last integration – this assures that the transients are integrated out. Start up AUTO by clicking `File Auto` (**F A**). Since there is only one parameter, you don't have to worry about setting the active parameters. From here on in, all commands will be in the **AUTO window** . Click on `Axes Hilo` and fill the dialog as follows:

Y-axis: X
Main Parm:: a
2nd Parm:: a
Xmin:: -.75
Ymin:: -3
Xmax:: .75
Ymax:: 3

and click **Ok** . We are plotting `X` versus the parameter `a` in the window $[-.75, .75] \times [-3, 3]$. Click on `Numerics` and set `Par min: -0.75` and `Par max:0.75`; then click on `OK` . Now we are ready to run. But have your mouse poised over the `ABORT` button since AUTO will run in circles until you stop it. (This is because the periodic solution is an isola – an isolated circle of solutions – and AUTO doesn't stop on such branches. When you are ready to run, click on `Run Periodic` . Click on `ABORT` if it looks like AUTO is running in circles. (Another way to prevent this is to set the total number of computed points, `Npts`, to a low value from the `Numerics` menu. However, there is a danger that you won't compute all the points on the diagram.) A pair of ellipses should appear and you should see something like the Figure 7.3. The filled dots represent the maximum and minimum values of the stable periodic solution and the unfilled dots are the max/min values of the unstable periodic branch. A better way to look at this is to view the norm. Click on `Axes Norm` and change `Ymin` to `0`. Then click on `OK` and `reDraw` . Use the `Grab` command and the arrow keys to move around the loop. Note that at $a = \pm 0.5$, there is a limit point of oscillations.

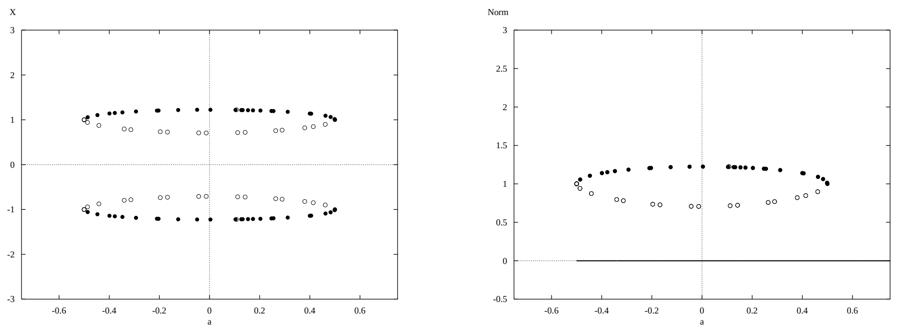


Figure 7.3. *The diagram for the isola example. The left figure shows the maximum and minimum of the stable (filled circles) and unstable (empty circles) limit cycles. The right figure shows the norm of the limit cycles and the stable fixed point at 0.*

XPPAUT lets you pick up other branches of solutions as and put them on the same diagram. Click on `File Clear grab.` This clears any grabbed points and thus allows you to start from a completely different point. From the `Axes Norm` menu, change `Ymin=-.5`. Now, go back into the **Main Window** of *XPPAUT* . Change a to -.5 and change the initial conditions to `X=0` and `Y=0`. This is a fixed point. Go back to the **AUTO window** and click on `Run Steady state` . When prompted to destroy the diagram, click on `No` . This way, you don't delete the part of the diagram that was already computed. You should see a thick horizontal line appear indicating the existence of a stable fixed point. You should see something like the right panel of Figure 7.3. If you want a postscript printout of this, click on `File Postscript` and save it with a name of your choice. Click on `File Reset diagram` to get rid of the diagram and all the AUTO junk left on your disk. (This is not necessary – these files are deleted when you exit *XPPAUT* , but it is always a good idea.) Resetting a diagram is also useful if you want to make changes in the parameters and start fresh.

7.1.2 A "real" example

The examples presented above are pretty lame – we knew the answers already. Instead, lets analyze a real world example that we have already studied; the Morris-Lecar equations:

$$\begin{aligned} C\frac{dV}{dt} &= I + g_l(E_l - V) + g_k w(E_K - V) + g_{Ca} m_\infty(V)(E_{Ca} - V) \\ \frac{dw}{dt} &= (w_\infty(V) - w)\lambda_w(V). \end{aligned}$$

We considered a dimensionless version of these and will continue to do so here. Here is the ODE file for a single neuron, `ml.ode`

```
# ml.ode
# morris-lecar; dimensionless
v'=I+gl*(el-v)+gk*w*(ek-v)+gca*minf(v)*(eca-v)
w'=(winf(v)-w)*lamw(v)
par I=0,phi=.333
par ek=-.7,eca=1,el=-.5
par gl=.5,gk=2,gca=1
par v1=-.01,v2=0.15,v3=0.1,v4=0.145
minf(v)=.5*(1+tanh((v-v1)/v2))
winf(v)=.5*(1+tanh((v-v3)/v4))
lamw(v)= phi*cosh((v-v3)/(2*v4))
aux ica=gca*minf(v)*(v-eca)
aux ik=gk*w*(v-ek)
@ total=50,xlo=-.6,xhi=.5,ylo=-.25,yhi=.75
@ xplot=v,yplot=w
done
```

Note that the first two parameters, the current `I` and the overall rate of the potassium current `phi` are the ones we want to vary. (I have added a pair of auxiliary variables which track the active currents). Start up *XPPAUT* and integrate this equation using `Initconds Go` followed by `Initconds Last` to get rid of all the transients. Click on `File Auto` to bring up AUTO. Since the parameters that we want to vary were the first defined, they will be among the list of usable parameters. Click on `Parameter` in the **AUTO window** just to check and see that `I` and `phi` are the first two entries. Click on `Ok` to close the dialog box. From here on in, the rest of the commands will be in the **AUTO window** . Click on `Axes Hilo` to set up the graphics axes. Fill in the dialog box as follows and then close it:

`Y-axis:` V
`Main Parm::` I
`2nd Parm::` phi
`Xmin::` -.1
`Ymin::` -.5
`Xmax::` .5
`Ymax::` .5

Next open the `Numerics` dialog box, change `Par Min` to `-.5` and `Dsmax` to `0.05` close it. Now you should be OK to run. Click on `Run Steady state` . You should see a cubic like curve with a few special points on it (see figure 7.4). Click on `Grab` and move through the diagram. You will see limit points (LP) at $I = 0.08326$ and $I = -0.02072$ as well as a Hopf bifurcation point (HB) at $I = 0.20415$. One expects to pick up a branch of periodic solutions at a Hopf bifurcation. When you get to this point, click **Enter** to grab it. Now, you can try to compute the periodic orbit emanating from this point. Click on `Run` and you will get a list of choices. Choose `Periodic` and when you see a sort of smudge at the end of the branch of limit cycles, click on `ABORT` to stop AUTO. You should see something like Figure 7.4. The "smudge" is due to AUTO's difficulty at computing the long period oscillation which terminates on the lower fold point. This shows a number of interesting features that are quite important in bifurcation theory. First, note that the branch of periodic orbits that comes out of the Hopf point is unstable (open circles) and then it turns around at a limit point of oscillations at $I = 0.242$ so that a stable and unstable limit cycle coalesce here. Thus, for a small range of currents, there is bistability between the upper fixed point and the stable limit cycle. The stable limit cycle (solid circles) terminates on the fold giving rise to a SNIC (saddle-node infinite period cycle). In fact, the frequency of the limit-cycle goes to zero as the saddle-node is approached. To see this, click on `Axes` `Frequency` , fill in the dialog as follows and close it:

`Y-axis:` V
`Main Parm::` I
`2nd Parm::` phi
`Xmin::` .05
`Ymin::` 0
`Xmax::` .3
`Ymax::` .2

The lower branch looks just like a square-root which is what the theory of this type of bifurcation predicts. Figure 7.4 shows the frequency of both the stable and unstable limit cycles.

We will now look at the two-parameter diagram for this model. Click on `Axes` `Two par` , fill in the dialog box as follows and close it:

`Y-axis:` V
`Main Parm::` I
`2nd Parm::` phi
`Xmin::` -.1
`Ymin::` 0
`Xmax::` .3
`Ymax::` 3

This creates a graph with `I` along the horizontal and `phi` along the vertical axis. Now click on `Grab` and hit the `Tab` key until you come to the first limit point (LP) which should be labeled 2. (Look at the bottom of the window as you move through the diagram until the desired point is found. Press `Enter` to grab it.) Click on `Run` and a vertical line will appear. The line is vertical because the limit point is independent of the parameter `phi` *which has no effect on the value of the fixed point* but can affect the stability. Again, click on `Grab` and `Tab` to the second limit point (LP, labeled 3). Click on `Run` again and you will see another vertical line corresponding to the left-most limit point. Within the two vertical lines, there are 3 fixed points and outside, there is one. Next, click on `Grab` and `Tab` to the Hopf point (HB, labeled 4). Click on `Run` `Two param` and you will see a diagonal line go down and right. This is the curve of Hopf bifurcation points. Finally, we want to extend the curve of Hopf points to the left. To do this, we must tell AUTO to

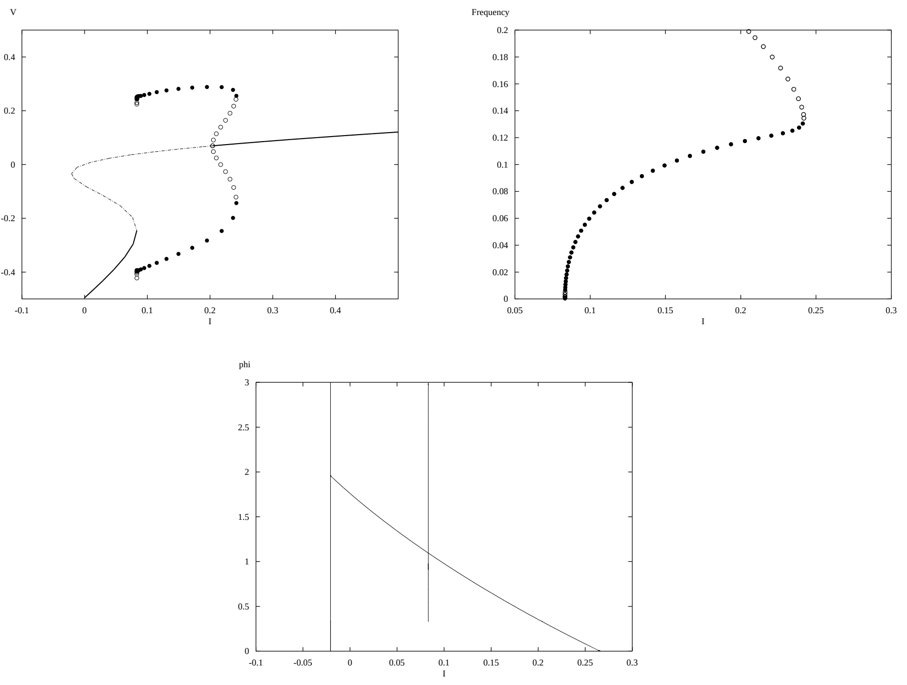


Figure 7.4. *Various diagrams associated with the Morris-Lecar equation. Top left: One-parameter diagram showing fixed points and limit cycles. Top right: Frequency of the periodic orbits. Bottom: Two-parameter diagram showing the curve of Hopf points and (straight-line) the lines of saddle-node points. These are independent of the parameter,* `phi`*.*

reverse direction. Click on `Numerics` and change `Ds=0.02` to `Ds=-0.02` and close the dialog box. Recall that AUTO uses the sign of `Ds` to determine a direction to go. Finally, click on `Grab` and get that Hopf bifurcation point again. Click on `Run` `Two param` and you will see the curve go up and to the left. It crosses the right line of fold points and terminates on the left line. The intersection with the right line of fold points is irrelevant since the Hopf occurs on the upper branch of solutions and the right fold line represents the loss of the lower branch of fixed points. The termination of the curve of Hopf points on the left line of folds signifies a new higher-order bifurcation point – the Takens-Bogdanov bifurcation. There is a double zero eigenvalue; the Hopf bifurcation and the saddle-node have coalesced. This bifurcation often signals the presence of homoclinic orbits. For `phi` below this curve and between the two vertical lines, the upper branch of fixed points is unstable. Outside the vertical lines, there is only one fixed point. On the right, it is stable above the Hopf curve and unstable below. On the left, the fixed point is always stable. For example, choosing `I=.15 phi=.5` is in the regime where there is one fixed point and it is unstable. Since you can readily prove that all solutions stay bounded for this model, this implies that there must be at least one stable periodic solution. Look at the phaseplane in this case and verify that you are right.

Here are some hints for computing the complete diagram for a system:

- **ALWAYS** make sure you are starting at a fixed point or clearly defined limit cycle. For fixed points, click on `Initialconds Go` and then `Initialconds Last` a few time to get rid of transients. For limit cycles, get a good estimate of the period and integrate **one** full period and no more or less.
- To navigate the diagram quickly, click on `Tab` as this just jumps to special point; if you get lost in the diagram, use the `Axes Fit reDraw` sequence to put the entire diagram on the page. Or if there is a really complex part, use the `Axes Zoom` feature to magnify the given area.
- Follow all branch points; AUTO will generally go back to all branches of fixed points and try to follow these automatically. However, for limit cycles which have branch points such as a pitchfork bifurcation, AUTO does not automatically compute them. Use Grab to move to such points, labeled `BP` and then click on `Run` . Change the sign of `Ds` in the numerics dialog the branch off in a different direction.
- Grab all Hopf points and try to find the periodic solutions that arises from them.
- For two-parameter bifurcations, also change the sign of `Ds` to go in both directions.
- If AUTO fails from the beginning – as indicated by the `MX` label – then you haven't given it a proper starting value. Make sure you are on a good periodic orbit or at a true fixed point or have a true solution to the boundary value problem.
- If AUTO fails after partially completing the diagram and you would like to change numerical parameters and have another go, destroy the diagram first before continuing: `Grab` the initial starting point and `File Reset diagram` to destroy the diagram.
- If AUTO seems to not be able to continue, change `dsmin` to make it smaller; for periodic orbits and boundary-value problems, change `ntst` and make it larger.
- If AUTO seems to miss a bifurcation point that is clearly there, clear the diagram, make `dsmax` smaller so that AUTO doesn't miss it. Reset the diagram and recompute.

7.1.3 Exercises

1. Set `phi=1.2` and compute the one-parameter bifurcation for `ml.ode` diagram using I as the parameter. Make sure that `Par Min` is `-0.5`. Note how the periodic orbit intersects the middle branch of fixed points at a homoclinic orbit. Set I to be near this intersection and look at the phase-plane of the Morris-Lecar model. See if you can understand the global dynamics for this value of I. Next observe that for $I = 0.072$ the system appears to be tri-

stable. There are two stable fixed points and one stable periodic orbit. Draw the complete phase-portrait with $I = 0.072$. Include unstable periodics which you can compute by integrating backwards.

2. Compute the bifurcation diagram using the current as a parameter for the dimensionless Morris-Lecar equations using the parameter values in the above example, but setting, `v3=.0166, v4=.25, phi=.333` and looking in the window $-0.5 < v < 0.5$ and $0 < I < 0.5$. Make sure that you start at a fixed point when $I = 0$. Compute the branch(es) of periodic solutions emanating from the Hopf bifurcation. Plot the frequency of the periodic orbits as a function of the current, I. For a bonus problem, compute the two parameter curve of Hopf bifurcation points using `phi` as the second parameter – window the two-parameter graph with `0 < phi < 2`. Thus show that if $\phi < \phi^*$ there are two Hopf bifurcations. Find ϕ^*.

3. Compute the one-parameter bifurcation diagram for the Brusselator:

$$u' = a - (b+1)u - vu^2 \quad v' = bu - vu^2$$

with $a = 1, b = 1.5, u = 1, v = 1.5$ and treating b as the bifurcation parameter. In `Numerics` set `Par max` to be 4. Window the diagram $0 < u < 6$ and $0 < b < 4$.

4. This one is for bifurcation jockey's only. Explore the bifurcation diagram of the coupled Brusselators:

$$u_1' = f(u_1, v_1) \tag{7.4}$$
$$v_1' = g(u_1, v_1) + d(v_2 - v_1) \tag{7.5}$$
$$u_2' = f(u_2, v_2) \tag{7.6}$$
$$v_2' = g(u_2, v_2) + d(v_1 - v_2) \tag{7.7}$$
$$f(u, v) = a - (b+1)u + vu^2 \tag{7.8}$$
$$g(u, v) = bu - vu^2 \tag{7.9}$$

with $d = 0.2, a = 1$ and letting b be the bifurcation parameter. In order to do this one completely, you should follow all special points. In particular, branch points of periodic orbits, labeled `BP`. For these, you may have to grab them twice and change the direction of the continuation (`Ds`) in the `Numerics` menu. A number of other parameters have to be changed in the `Numerics` menu for AUTO, namely the number of collocation points used to track periodic orbits, `Ntst=30`, the minimum step size `Dsmin=1e-5` and the maximum step size, `Dsmax=0.1`. The diagram should be drawn with $0 < b < 5$ and $0 < u_1 < 7$. As with many internal parameters, most of the AUTO parameters can be set with options in the ODE file. Here is my version of the above equations

```
# bruss2.ode
# two Brusselator equations
```

```
f(u,v)=a-(b+1)*u+v*u^2
g(u,v)=b*u-v*u^2
u1'=f(u1,v1)+du*(u2-u1)
v1'=g(u1,v1)+dv*(v2-v1)
u2'=f(u2,v2)+du*(u1-u2)
v2'=g(u2,v2)+dv*(v1-v2)
par b=1.5,a=1,du=0,dv=.2
init u1=1,u2=1,v1=1.5,v2=1.5
@ ntst=30,dsmin=1e-5,dsmax=0.1,parmin=0,parmax=5,autoxmin=0
@ autoxmax=5,autoymin=0,autoymax=7
done
```

Note that a few years ago, this computation and some discussion of the resultant system would have been considered a decent research problem. Now it's an exercise in a book.

7.2 Maps, boundary value problems, and forced systems.

XPPAUT allows you to study continuation of discrete dynamical systems, boundary value problems, and with some preparation, periodically forced systems. The use of AUTO with maps is rather restricted since all one can do is follow branch points and two parameter continuations of limit-points and Neimark-Sacker points. (Recall the Neimark-Sacker bifurcation is the discrete analogue of the Hopf bifurcation.)

7.2.1 Maps

XPPAUT makes some aspects of tracking the behavior of maps pretty easy, mainly because it allows you to study the continuation of $f^k(x)$, the k^{th} iterate of the map, rather than just the map. For, example, consider the logistic map:

$$x_{n+1} = ax_n(1-x_n) \equiv f(x,a)$$

with parameter a. If we were to run this with AUTO, it would say that at $a=3$ there was a Hopf bifurcation with period 2. Since this is really a period-doubling bifurcation, we could try to follow it. However, AUTO cannot continue from "Hopf" points of maps. Thus, how could we follow this apparent period 2 point. Since this period two point arises from the primary branch of solutions, if we look at $f^2(x,a) = f(f(x,a),a)$, then the period two orbit is just a branch of the second iterate and AUTO will follow it. Thus, if we wanted to find all the periodic orbits dividing 8, we could just look at the 8th iterate of the map with a as the parameter. Here is our version of the logistic map:

```
# logisticmap.ode
# the classic logistic map
x(t+1)=a*x*(1-x)
```

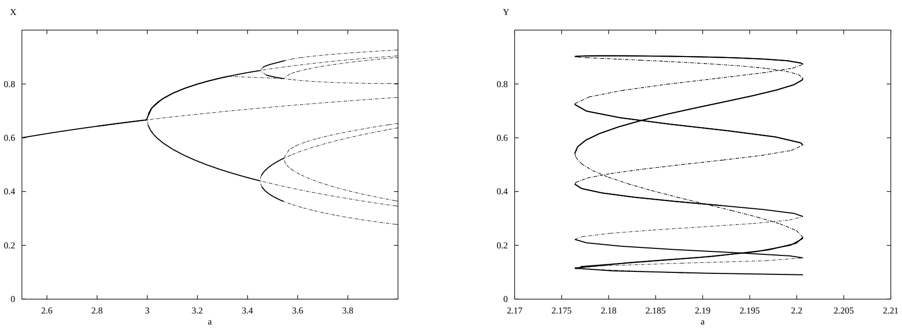


Figure 7.5. *Bifurcation diagram for period 8 orbits of the logistic map (left) and period 7 orbits for the delayed logistic map (right)*

```
par a=2.8
init x=.64285
@ total=200,meth=disc
done
```

Run *XPPAUT* with this. We have started pretty close to the fixed point. Click on `nUmerics` in the main *XPPAUT* window and change `nOutput` from 1 to 8 and exit the numerics menu. When AUTO is run within *XPPAUT* , it looks at the numeric parameter `nOut` and uses this to iterate the map `nOut` times before returning the value of the map. Thus, it is possible to use *XPPAUT* to continue branches of periodic points of maps rather easily. We will thus look at the 8th iterate of the map. Fire up AUTO `File Auto` . Click on `Axes` can choose `Hi Lo` . Fill in the dialog box as follows and then click `Ok` .

`Y-axis:` X
`Main Parm:` a
`2nd Parm:` a
`Xmin:` 2.5
`Ymin:` 0
`Xmax:` 4
`Ymax:` 1

Click on `Numerics` in the AUTO window and change `Dsmax` from 0.5 to 0.05 and change `Par max` from 2 to 4. Now click on `Run` and you will see a rather messy looking diagram. There are three distinct branch points at $a = 3, 3.449, 3.544$ corresponding to the period 2, 4 and 8 bifurcations. You should see a figure like Fig 7.5.

We can use a similar trick to find a set of period 7 orbits in a two-dimensional map, the "delayed logistic" map:

$$x_{n+1} = y_n \quad y_{n+1} = ax_n(1 - y_n) + \epsilon$$

Here, there is an additional parameter, ϵ which will play no role in our analysis. At $a = 2.177$ there is a period 7 orbit that starts at $(x, y) = (.1115, .12111)$. We will let AUTO find the entire branch of solutions for this model. Here is the ODE file:

```
# delayed logistic map
# illustrates a period 7 continuation for the delayed logistic map
y'=x
x'=a*x*(1-y)+eps
par a=2.177,eps=0
init x=.1115,y=.12111
@ meth=discrete,total=100,nout=7
@ autoxmin=2.17,autoxmax=2.21,autoymin=0,autoymax=1
@ dsmax=.025,dsmin=.00001,parmin=2,parmax=2.5
done
```

I have taken the liberty of setting some of the AUTO parameters so that you can just run this with no hassle. I have also set `nout=7` so that AUTO looks at the 7th iterate of the map. Start up *XPPAUT* and iterate it for a few times to make sure you have no transients. Then, click on `File Auto` and in the **AUTO window** , click on `Run` to compute the diagram. It looks complicated but in reality, it simply shows that for $2.1764 < a < 2.20$ there are two period 7 points, one is stable and the other is unstable. The reason there appears to be many branches is that for a period 7 point, there are 7 possible starting values. Figure 7.5 shows the diagram.

We will now do a two-parameter bifurcation analysis of the above problem. You can either edit all the parameters in the above file, or simply load in the stripped down version:

```
# del_log2.ode
# delayed logistic map
# simple 2 parameter stuff
y'=x
x'=a*x*(1-y)+eps
par a=1.5,eps=0
init x=.333,y=.333
@ meth=discrete,total=100
done
```

Run this model. Then start up AUTO. Use the `Axes Hi Lo` and change them so the `Xmin=0, Ymin=-1, Xmax=4, Ymax=1`. In AUTO `Numerics` , change `Par max` to 4. Then run the continuation. You should see a so-called Hopf point labeled 2. Note that it says the period is 6. This may mean that there is a period 6 point lurking nearby.

Let's draw the two parameter diagram using this heretofore worthless parameter ϵ. Grab the Hopf point by clicking on `Grab` , moving to it, and then tapping `Enter` . Then `Axes Two par` and click on `Ok` in the dialog box – the window is fine. Click on `Run` and you will see a curve of Hopf points in the `eps-a` plane and another lower curve which is tangent to this curve at a point approximately $a = 2.8$

and $\epsilon = -0.3$. Along the lower curve, there is an eigenvalue of $+1$ and along the upper curve are eigenvalues on the unit circle but not real.

7.2.2 Exercises

1. The Ricker model is another discrete population model that is often used:

$$x_{n+1} = ax_n e^{-x_n}.$$

   Starting at $a = 1.2$ and $x = 0.18232$, compute the sequence of period doublings up to period 8 using the 8th iterate. The parameter a should be allowed to range between 0 and 20 and the population x lines between 0 and 8.

2. The modified Nicholson-Bailey system models a predator prey interaction:

$$x_{n+1} = x_n \exp(r(1 - x_n/k) - ay_n) \quad y_{n+1} = x_n(1 - \exp(-ay_n))$$

   where r, k, a are positive parameters. Start at $r = .5, k = .5, a = 1, x = 1, y = 0$ and create the bifurcation diagram for $0 < k < 5$. Note that k is the carrying capacity and represents the amount of prey that exists when there are no predators. Solve the system when $k = 3.2$ Does the resulting "periodic" solution have a period roughly like predicted from the Hopf point? Set $k = 5, r = 1.6$ and verify that there is a period 7 orbit starting at roughly $x = .376, y = .198$. Continue this period 7 orbit holding r fixed and letting $4.5 < k < 5.5$ be the bifurcation parameter. Plot the prey, x in a window $0 < x < 6$ and finally set `Dsmax=0.1.` You will get a nice corkscrew!

7.3 Boundary value problems

XPPAUT is able to solve boundary values problems as we saw earlier in the book. However, the method used by *XPPAUT* depends on shooting which can be wildly difficult when the desired evolution equation has large positive eigenvalues. Furthermore, *XPPAUT* cannot track solutions through turning points nor does it find branch points which are often indicators of non-trivial solutions. Thus, if you can write your problem as an autonomous boundary value problem on the interval $[0, 1]$ and find a simple starting solution, then AUTO is probably your best bet for tracking a solution. We will examine a few cases.

EXAMPLE 1. Here is an example form the original AUTO manual that involves branch switching:

$$u' = v \quad v' = -(\pi r)^2 u + u^2 \quad u(0) = u(1) = 0.$$

Here is an ODE file that has all the required AUTO parameters set for you:

```
# bvp1.ode
# using AUTO to find the branches of a BVP
```

```
par r=0
u'=v
v'=-(r*pi)^2*u+u^2
b u
b u'
@ total=1,dt=.01
@ dsmax=.2,ds=.2,autoxmin=0,autoxmax=6,autoymin=0,autoymax=100
@ ntst=5,parmax=5
@ meth=cvode,bound=10000,xhi=1,ylo=-50,yhi=50
done
```

Run *XPPAUT* and integrate this once to get the trivial solution loaded. Click on `File` `Auto` to get the AUTO window up. In the **AUTO window** , click on `Run` `Bndry Value` and you will see a series of labeled points along the bottom of the figure. Click on `Numerics` and change `Dsmax` to 10 and then click on `Ok` . `Grab` the first branch point (`BP`) (labeled 2). Then click on `Run` again; AUTO automatically switches when asked to continue from branch points. You will see a branch rising up from this point. When AUTO is done, grab the next branch point labeled 3 and continue this. Do this for the branch points at $r = 3, 4$ as well. Now, click on `Numerics` and change `Ds` to -0.2 to change directions click on `Ok` . Grab each of the branch points and run again. For some of them, it will appear like nothing has happened; this is because the maximum value of u is the same on each branch so that they are not distinguishable. Click on `Axes` `Hi` and change the `Y-axis` to `v` instead of `u`, click on `Ok` and then `reDraw` and you will see a slightly different view.

TRICK: Distinguishing branches in AUTO. Here is a trick that you can use to avoid the symmetry problem in graphing the curves of solutions. As we saw in this example, the things that AUTO plots – maximum of a solution and the norm of a solution – cannot always lead to distinctions. So here is a way to fool AUTO into letting you plot something else in the bifurcation diagram. For the above problem, the value of v at the origin is zero for the trivial branch of solutions, but for the solutions that emanate from each branch point it is non-zero and it is necessarily different. (The reason for this follows from the uniqueness theorem for ODEs. Since $u(0) = 0$, if $v_1(0) = v_2(0)$ are two different branches, then the solutions are identical.) Thus, we introduce a dummy differential equation

$$q' = 0$$

with a boundary condition $q(0) = v(0)$. Then the solution to this dummy equation satisfying the additional boundary condition is just $q(t) = v(0)$ for all t. Add the following two lines to the above ODE file:

```
q'=0
b q-v
```

The dimension of the system is a bit larger but this is a triviality. Figure 7.6 shows the result of this trick.

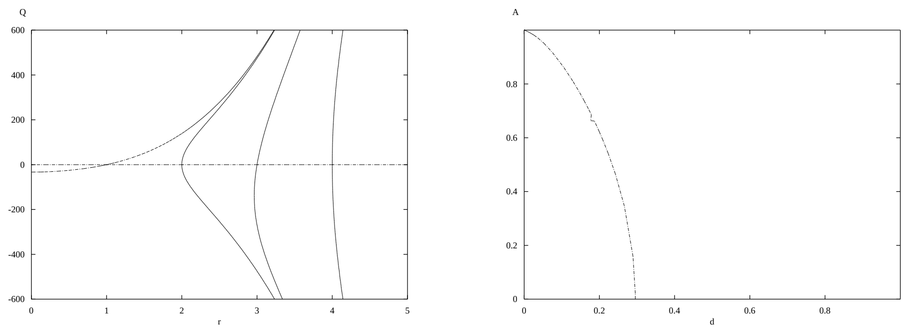


Figure 7.6. *Solution to the boundary value problems from example 1 (left) and example 2 (right).*

EXAMPLE 2. Here we revisit the example studied in chapter 4 which arises in the analysis of a reaction-diffusion equation on a disk. Recall that the boundary value problem is:

$$0 = A(1-A^2) + d(A'' - (A/r)' - Ak^2)$$
$$\Omega = qA^2 + d(k' + k/r - 2kA'/A),$$

with boundary conditions, $A'(1) = k(1) = 0$, and

$$\lim_{r\to 0} \frac{A(r)}{r} = A'(0) < \infty \quad \lim_{r\to 0} \frac{k(r)}{r} < \infty.$$

We handled the behavior as $r \to 0$ by forming a Taylor series near the origin. We let r_0 be close to zero and start the problem at $r = r_0$. This problem is nonautonomous, so we introduce the differential equation

$$r' = 1$$

with boundary condition $r(0) = r_0$. As before Ω is a free parameter so we write this as an ODE also. Here is the new ODE file which is autonomous and thus suitable for AUTO:

```
# gberg_auto.ode
#
init a=0.00118  ap=1.18  k=0  omeg=-0.19,r=.001
# domain is reasonably small sqrt(1/d)
par d=0.177  q=0.5, r0=.001
# the odes...
a'=ap
ap'=a*k*k-ap/r+a/(r*r)-a*(1-a*a)/d
k'=-k/r-2*k*ap/a-(omeg+q*a*a)/d
```

```
# extras to make it autonomous
omeg'=0
r'=1
# the boundary conditions
# at r=0
bndry  a-r*ap
bndry  k
bndry r-r0
# at r=1
bndry  ap'
bndry  k'
# set it up for the user
@ xhi=1.001,dt=.01,total=1.001,ylo=0
@ parmax=1,autoxmax=1,autoxmin=0,autoymax=1,autoymin=0
done
```

We have added a few setup parameters for AUTO. I have also computed a good starting point through trial and error. The parameter of interest is d which is like the diffusion constant; is d increases, the effective domain size decreases and as $d \to 0$, the domain becomes infinite. *XPPAUT* can only follow a branch with d as a parameter in a limited range, thus, we will use AUTO to get a fuller picture. Run *XPPAUT* with this ODE file and integrate it once to load the initial solution. Click on `File Auto` to get the AUTO window. In the **AUTO window** , click on `Run Bdry value` and then when the curve hits the horizontal axis, click on `ABORT` to stop it. Click on `Numerics` and change `Ds` to -0.02 and click `Ok` so that we can continue in the opposite direction. `Grab` the starting point (labeled 1). Run again and when the curve gets fairly close to the vertical axis, `ABORT` the calculation. You will see a parabola that intersects the horizontal axis at about $d = 0.295$. This tells us that if the diffusion is too great, it is impossible to support rotating waves. Similarly, as d gets smaller, the magnitude of the rotating waves increases; they seem to exist in arbitrarily large domains.

7.3.1 Homoclinics and heteroclinics

A recent version of AUTO includes a library of routines called HOMCONT which allow the user to track homoclinic and heteroclinic orbits. *XPPAUT* incorporates some aspects of this package. The hardest part of computing a branch of homoclinics is finding a starting point. Consider a differential equation:

$$x' = f(x, \alpha)$$

where α is a free parameter. Homoclinics are codimension one trajectories; that is, they are expected to occur only at a particular value of a parameter, say, $\alpha = 0$. We suppose that we have computed an approximate homoclinic to the fixed point $\bar{x}$ which has an n_s-dimensional stable manifold and an n_u-dimensional unstable manifold. We assume $n_s + n_u = n$ where n is the dimension of the system. The remaining discussion is based on Sandstede et al. The way that a homoclinic is computed is to approximate it on a finite interval; say $[0, P]$. We rescale time by $t = Ps$. We double the dimension of the system so that we can simultaneously solve for the equilibrium point as the parameters vary. We want to start along the unstable manifold and end on the stable manifold. Let L_u be the projection onto the unstable subspace of the linearization of f about the fixed point and let L_s be the projection onto the stable space. Then we want to solve the following system:

$$\begin{aligned} \frac{dx}{ds} &= Pf(x, \alpha) \\ \frac{dx_e}{ds} &= 0 \\ f(x_e(0)) &= 0 \\ L_s(x(0) - x_e(0)) &= 0 \\ L_u(x(1) - x_e(1)) &= 0 \end{aligned} \tag{7.10}$$

Note that there are $2n$ differential equations and $2n$ boundary conditions; n for the equilibrium, n_s at $s = 0$ and n_u at $s = 1$. There is one more condition required. Clearly one solution to this boundary value problem is $x(s) \equiv x_e(s) \equiv \bar{x}$ which is pretty useless. However, any translation in time of the homoclinic is also a homoclinic so we have to somehow define a phase of the homoclinic. Suppose that we have computed a homoclinic, $\hat{x}(s)$. Then we want to minimize the least-squares difference between the new solution and the old solution to set the phase. This leads to the following integral condition:

$$\int_0^1 \hat{x}'(s)(\hat{x}(s) - x(s))\, ds = 0. \tag{7.11}$$

This is *one* more condition which accounts for the need for an additional free parameter.

XPPAUT allows you to specify the projection boundary conditions and by setting a particular flag on in AUTO, you can implement the integral condition. Since the *XPPAUT* version of AUTO does not allow you to have more conditions than there are differential equations, you should pick one parameter which will be slaved to all the other ones you vary and let this satisfy a trivial differential equation,

$$\alpha' = 0.$$

This will be varied by AUTO to satisfy the conditions (7.10,7.11).

Here is an example of continuing a homoclinic in two-dimensions.

$$x' = y \quad y' = x(1 - x) - ax + \sigma xy$$

When $(a, \sigma) = (0, 0)$ there is a homoclinic orbit (Prove this by integrating the equations; this is a conservative dynamical system.) For small a it is possible to prove that there is a homoclinic orbit for a particular choice of $\sigma(a)$ using Melnikov methods (see Guckenheimer and Holmes). We now write the equations as a 5-dimensional system using σ as the slaved parameter and introducing a parameter, P for the period:

$$\begin{aligned} x' &= Pf(x,y) \\ y' &= Pg(x,y) \\ x_e' &= 0 \\ y_e' &= 0 \\ \sigma' &= 0 \end{aligned}$$

where $f(x,y) = y$, $g(x,y) = x(1-x) - ax + \sigma xy$ and the following boundary conditions

$$\begin{aligned} 0 &= f(x_e, y_e) \\ 0 &= g(x_e, y_e) \\ 0 &= L_s(x(0) - x_e, y(0) - y_e) \\ 0 &= L_u(x(1) - x_e, y(1) - y_e) \end{aligned}$$

and the integral condition. *XPPAUT* has a defined function for the projection boundary conditions called `hom_bcs(k)` where `k=0,1,...,n-1` corresponding to the total number required. You do not need to be concerned with ordering at this point as long as you get them all and you give XPP the required information. Here is the ODE file:

```
# tsthomi.ode
f(x,y)=y
g(x,y)=x*(1-x)-a*y+sig*x*y
x'=f(x,y)*per
y'=g(x,y)*per
# auxiliary ODE for fixed point
xe'=0
ye'=0
#
# free parameter
sig'=0
# (xe,ye) are the fixed points at the ends
b f(xe,ye)
b g(xe,ye)
# project off the fixed point from unstable manifold
b hom_bcs(0)
# project onto the stable manifold
b hom_bcs(1)
par per=8.1,a=0
init x=.1,y=.1
@ total=1.01,meth=8,dt=.001
@ xlo=-.2,xhi=1.6,ylo=-1,yhi=1,xp=x,yp=y
done
```

The only new feature is the projection conditions. *XPPAUT's boundary value solver will not work here since there are more equations than conditions and it doesn't know about the integral condition.* I have set the total integration time to 1 and have added the additional parameter `per` corresponding to the parameter P in the differential equation. I use the Dormand-Prince order 8 integrator as it is pretty accurate. I have also set the view to be the $(x, y)-$plane. Note that this is a pretty rough approximation of the true homoclinic. We will use AUTO to improve this before continuing in the parameter a. Run *XPPAUT* with this ODE file and integrate the equations. You will get a rough homoclinic pretty far from the fixed point. Click on `File Auto` to the the **AUTO window**. Now click on `Axes Hi` . Choose `xmin=0,xmax=50,ymin=-6,ymax=6` and also select `sig` as the variable in the y-axis. Click on `OK` and bring up the **AUTO window** `Numerics` dialog. Change `Ntst=35, Dsmin=1e-4,Dsmax=5`, `Par Max=50,EPSL=EPSU=EPSS=1e-7` and click `OK` . Now, before you run the program, click on `Usr Period` and choose `3` for the number. We want AUTO to output at particular values of the parameter `per` corresponding to P. When the dialog comes up, fill the first three entries in as `per=20,per=35,per=50` respectively and click `OK` . This forces AUTO to output when P reaches these three values. Now, click on `Run` and choose `Homoclinic` . A little dialog box appears. Fill it in as follows and click OK:

`Left Eq:` Xe
`Right Eq:` Xe
`NUnstable:` 1
`NStable:` 1

You must tell AUTO the dimension of the stable and unstable manifolds as well as the fixed point to which the orbit is homoclinic. (Note that if you ever fill this in wrong or need to change it, you can access it from the **Main Window** menu under `Bndry Value Homoclinic` .) Once you click on `OK` , you should see a straight line across the screen as the homoclinic approximation gets better. Click on `Grab` and grab the third point corresponding to the point `Per=35`. For fun, in the **Main Window** , click on `Initial Conds Go` and you will see a much better homoclinic orbit.

Now that we have a much improved homoclinic orbit, we will continue in the parameter a as desired. First, let's make sure we get the orbits when $a = -6, -4, -2, 2, 4, 6$. We will click on `Usr Period` and choose `6` . Type in the following in the first six entries: `a=-6,a=-4,a=-2,a=2,a=4,a=6` and click `Ok` . Click on `Axes Hi` to change the axes and the continuation parameter. Change the `Main Parm` to `a`, `Xmin=-7,Xmax=7` and click `OK` . Click on `Numerics` and change `Par Min=-6, Par Max=6` and then click `OK` . Now click on `Run` and you will see a line that is almost diagonal. When done, click on `Grab` again, and watch the bottom of the AUTO window until you see `Per=35` and click Enter. In the **AUTO window** `Numerics` menu, change `Ds=-.02` to change directions, and click `Ok` . Now click `Run` and there will be another diagonal line that is in the opposite direction. Click

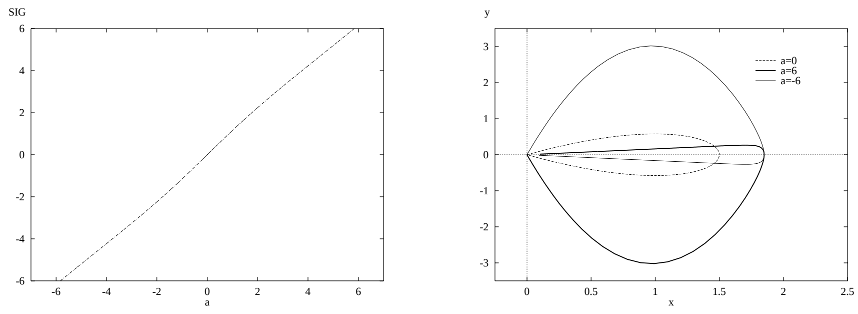


Figure 7.7. *Left: The continuation of the homoclinic orbit for the example problem; Right: Sample orbits computed for $a = 0, \pm 6$ in the (x, y) phaseplane.*

on `Grab` and grab point number 7 corresponding to `a=6`. In the **Main Window** , click on `Init Conds` `Go` and you will see a distorted homoclinic. It is not that great and could be improved probably by continuing with `Per` some more. Grab the point labeled 11 (`a=-6`) and the **Main Window** try to integrate it. It doesn't look even close. This is because the homoclinic orbit is unstable and shooting (which is what we are doing when we integrate the equation) is extremely sensitive to the stability of the orbits. In the **AUTO window** , click on `File` `Import Orbit` to get the orbit that AUTO computed using collocation. In the **Main Window** , click on `Restore` and you will see a much better version of the homoclinic orbit. This is because collocation methods are not sensitive to the stability of orbits! In fact, you can verify that the fixed point (0,0) is a saddle-point with a positive eigenvalue, λ_u and a negative one of λ_s whose sum is the trace of the linearized matrix, $-a$. The sum of the eigenvalues is called the **saddle-quantity** and if it is positive (for us, $a < 0$), then the homoclinic is unstable. Figure 7.7 shows the continuation and some representative orbits.

Heteroclinics We now describe how to find heteroclinic orbits. The methods are the same except that we must track two *different* fixed points. Thus, we need an additional n equations for the other fixed point. As with homoclinic orbits, we go from the unstable manifold to the stable manifold. In this case, the "left" fixed point is the one emerging from the unstable manifold and the "right" fixed point is the one going into the stable manifold. Thus, the dynamical system is :

$$\begin{aligned}
\frac{dx}{ds} &= Pf(x, \alpha) \\
\frac{dx_{left}}{ds} &= 0 \\
\frac{dx_{right}}{ds} &= 0 \\
f(x_{left}(0)) &= 0
\end{aligned}$$

$$\begin{aligned} f(x_{right}(1)) &= 0 \\ L_s(x(0) - x_{left}(0)) &= 0 \\ L_u(x(1) - x_{right}(1)) &= 0. \end{aligned}$$

The only difference is that we have the additional n equations for the right fixed point and the n additional boundary conditions. It is important that you give good values for the initial conditions for the two fixed points since they are different and you need to converge to them. The classic bistable reaction-diffusion equation provides a nice example of a heteroclinic. We examined this equation (6.1) in the previous chapter both by integrating the discretized partial differential equation and by shooting in the ODE arising from the traveling wave assumption. The equations are:

$$-cu' = u'' + u(1-u)(u-a)$$

which we rewrite as a system:

$$\begin{aligned} u' &= u_p \equiv f(u, u_p) \\ u_p' &= -cu_p - u(1-u)(u-a) \equiv g(u, u_p) \end{aligned}$$

The fixed point $(1, 0)$ has a one-dimensional unstable manifold and $(0, 0)$ as a one-dimensional stable manifold. Recall that we found that when $a = 0.25$, the velocity, c was approximately $c \approx 0.34375$. We will now use a different method to compute the solution – solve the approximate boundary-value problem. We seek a solution heteroclinic from $(1, 0)$ to $(0, 0)$. For $a = 0.5$ and $c = 0$, there is an exact solution joining the two saddle points. (Prove this by showing that $u_p^2 + u^4/2 - 2u^3/3 + u^4/2$ is constant along solutions when $a = 0.5, c = 0$.) We will use this as a starting point in our calculation. Here is the ODE file:

```
# tstheti.ode
# a heteroclinic orbit
# unstable at u=1, stable at u=0
f(u,up)=up
g(u,up)=-c*up-u*(1-u)*(u-a)
# the dynamics
u'=f(u,up)*per
up'=g(u,up)*per
# dummy equations for the fixed points
uleft'=0
upleft'=0
uright'=0
upright'=0
# the velocity parameter
c'=0
# fixed points
b f(uleft,upleft)
b g(uleft,upleft)
b f(uright,upright)
```

```
b g(uright,upright)
# projection conditions
b hom_bcs(0)
b hom_bcs(1)
# parameters
par per=6.67,a=.5
# initial data
init u=.918,up=-.0577,c=0
# initial fixed points
init uleft=1,upleft=0,uright=0,upright=0
@ total=1.01,dt=.01
@ xp=u,yp=up,xlo=-.25,xhi=1.25,ylo=-.75,yhi=.25
# some AUTO parameters
@ epss=1e-7,epsu=1e-7,epsl=1e-7,parmax=60,dsmax=5,dsmin=1e-4,ntst=35
done
```

I have added a few AUTO numerical settings so that I don't have to set them later. Run *XPPAUT* with this ODE file and integrate the equations. We will now continue this approximate heteroclinic in the parameter `per`. Fire up AUTO (`File Auto`) and click on `Axes Hi` . Put `c` on the y-axis and make `Xmin=0, Xmax=60, Ymin=-2, Ymax=2`. Then click OK. As above, we will also keep solutions at particular values of `per` by clicking on `Usr period 3` , choosing `per=20,per=40,per=60`, and then `OK` . Now we are ready to run. Click on `Run Homoclinic` . When the dialog box comes up set the following:

`Left Eq:` uleft
`Right Eq:` uright
`NUnstable:` 1
`NStable:` 1

and then click `OK` . You should see a nice straight line go across the screen. `Grab` the point labeled `per=40` and then click on `File Import Orbit` . Look at it in the **Main Window** by clicking `Restore` and freeze it. Now in the **AUTO window** , click on `Axes Hi` and change the `Main Parm` to `a` and `Xmax=1`. Click `OK` and then click on `Numerics` to get the Auto Numerics dialog. Change `Dsmax=0.1, Par Max=1` and click `OK` . Now click on `Usr Period 4` and make the four user functions `a=.75,a=.9,a=.25,a=.1` and click `OK` . Now click on `Run` and watch a line drawn across the screen. This is the velocity, c as a function of the threshold, a. Click on `Grab` and looking at the bottom of the screen, wait until you see `per=40` and then click on `Enter` . Now, open the `Numerics` dialog box and change `Ds=-.02` to go the other direction. Click on `Run` and you should see the rest of the line drawn across the screen. Click on `Grab` and move to the point labeled `a=0.25` Click on `File Import Orbit` and plot this in the **AUTO window**. Amazingly, there is essentially no difference in the two plots! Can you explain why? Notice the value of c at the point $a = .25$. It is very close to the value we computed by shooting in chapter 6.

7.3.2 Exercises

1. A nonlinear PDE that arises in the study of pattern formation has the form:

$$u_t = -u_{xxxx}/\pi^4 - au_{xx}/\pi^2 - u + r\tanh(u)$$

with boundary conditions

$$u(0,t) = u_{xx}(0,t) = u(1,t) = u_{xx}(1,t) = 0.$$

One steady state solution to this is $u = 0$. If a is large enough, the zero state loses stability as r increases and the resulting solution is a stationary spatial pattern. Since this solution branches fro the origin, we can use AUTO to try to find a non-trivial branch of solutions. The steady state solution is a fourth order nonlinear problem

$$u'''' = \pi^4(r\tanh(u) - u + au''/\pi^2)$$

with

$$u(0) = u''(0) = u(1) = u''(1) = 0.$$

We can rewrite this as a four-dimensional system and then write an ODE file for it. Here is a possible ODE file where I have set up AUTO for you as well:

```
# boundary value problem
# 0 = - (u_xxxx/pi^4+au_xx/pi^2)-u+r tanh(u)
# u,u_xx = 0 at x=0,1
u'=up
up'=upp
upp'=uppp
uppp'=pi^4*(r*tanh(u)-u-a*upp/(pi*pi))
par r=1,a=2
@ total=1.01,xhi=1,meth=cvode,bound=1000
@ dsmax=2,parmax=20,autoxmin=0,autoxmax=20
@ autoymin=-1,autoymax=4
b u
b u'
b upp
b upp'
done
```

Run this in *XPPAUT* , integrate once. Then look at the bifurcation diagram with r as the parameter. Follow any branches of solutions that arise. Grab a point on a nontrivial branch (not too far away from the branch point) and back in the **Main Window** of *XPPAUT* , integrate the solution showing that it is nonconstant.

2. Here is another nonlinear boundary value problem:

$$u'' = -r\exp(u), \quad u(0) = u(1) = 0$$

where r is a parameter. Write this as a pair of first order equations

$$u' = v \quad v' = -r\exp(u).$$

Starting with $r = 0$ use AUTO to continue the solution $u = 0$ for $0 < r < 10$. Window the `Y-axis` in the bifurcation diagram between 0 and 20. Show that if $0 < r < r^*$ there are two solutions to the boundary-value problem and if $r > r^*$ there appear to be none.

3. Consider the nonautonomous equation

$$u''(x) = -r\exp(ux^2) \quad 0 < x < 1 \quad u(1) = u(0) = 0.$$

By introducing the additional equation $x' = 1, x(0) = 0$ solve this using AUTO over the interval $0 \le r \le 20$.

4. Use the file `ml.ode` that we studied exercise 1 in the "real example" section, find the homoclinic orbit, and continue this using the two parameters `I,phi`. Do this by adding an equation for the applied current:

$$I' = 0$$

and treating `phi` as a parameter. You should start at the long period point that terminates the branch of periodic orbits. This is a hard exercise.

7.4 Periodically forced equations.

To solve periodically forced equations, one can always use the same trick we used in the previous section and increase the dimension by one. E.g

$$x' = f(x,t)$$

where f is periodic in t with period 1. We want to look for periodic solutions, so we solve

$$x' = f(x,s) \quad s' = 1 \quad x(0) - x(1) = 0 \quad s(0) = 1.$$

However, this method does not say anything about the stability of the solutions. Instead, what I do is introduce the two-dimensional dynamical system

$$u' = u(1 - u^2 - v^2) - \omega v \quad v' = v(1 - u^2 - v^2) + \omega u.$$

This admits an asymptotically stable periodic orbit, $(u(t), v(t)) = (\cos\omega t + \phi, \sin(\omega t + \phi))$ where ϕ is an arbitrary phase-shift. Thus, to solve a periodically driven system, I merely append this new system and couple $u(t)$ to it. Here is an example of a periodically forced bistable system

$$x' = x(1-x)(x-a) + c\cos\omega t.$$

For use with AUTO, we create the following ODE file

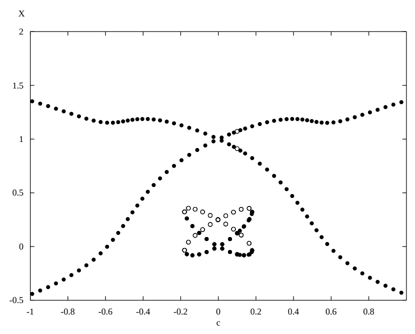


Figure 7.8. *Bifurcation diagram for the periodically driven bistable system*

```
# bistabfor.ode
# periodically forced bistable system
x'=x*(1-x)*(x-a)+c*u
u'=u*(1-u^2-v^2)-w*v
v'=v*(1-u^2-v^2)+w*u
init u=1,x=.043
par c=.1,a=.25,w=1
@ dt=.01,total=6.281,xlo=0,xhi=6.5
@ dsmax=.05,parmin=-1,parmax=1
@ autoxmax=1,autoxmin=-1
@ autoymin=-.5,autoymax=2
done
```

I have set it up for AUTO. Run *XPPAUT* with this file. Integrate the equations once. Now click on `File Auto` to get AUTO started. The diagram is set up so that the forcing strength, c is the main parameter. Click on `Run Periodic` to pick up the periodic solution. Click on `ABORT` to stop it from infinitely looping. You should see a little four-leafed clover pattern. Remember that this is showing the maximum and minimum of each periodic solution. There are two such solutions; one is stable and one is unstable. Click on `Grab` to load the first point. Click on `Enter` right afterwards. Now in the **AUTO window** click on `File Clear grab` . This makes AUTO forget the point it just grabbed. However, the point is still loaded into *XPPAUT* . In the **Initial Data Window** of *XPPAUT* , change the initial value of x to 1. In the **Main Window** of *XPPAUT* , click on `Initialconds Go` and then when that is done, `Initialconds Last` . We now have another periodic solution where x is large and not near 0. Back in the **AUTO window** , click on `Run Periodic` and a new branch of solutions will appear. The point of clearing the AUTO grab point is that AUTO can start fresh from a new point. Grab the starting point of this most recent computation – this may take many taps on the `Tab` key– until the cross is on the first point in the upper branch. Change the sign of `Ds` in the `Numerics` dialog of the **AUTO window** . Click on `Run Extend` to get the rest of the branch. You should see something like Figure 7.8.

7.4.1 Exercises

1. Periodically forced oscillators lead to remarkably complex behavior. Here, we will look at the periodically driven Fitzhugh-Nagumo equations (see chapter 3). Here is the appropriate ODE file:

   ```
   # fhnfor.ode
   # Fitzhugh-Nagumo equations with sinusoidal forcing
   v'=I+v*(1-v)*(v-a) -w+c*u
   w'=eps*(v-gamma*w)
   u'=u*(1-u^2-z^2)-f*z
   z'=z*(1-u^2-z^2)+f*u
   init u=1,v=.0505,w=.1911
   par c=.05,f=.8
   par I=0.2,a=.1,eps=.1,gamma=.25
   @ xp=V,yp=w,xlo=-.25,xhi=1.25,ylo=-.5,yhi=1,total=7.86
   @ dsmax=.05,parmin=-.5,parmax=1
   @ autoxmax=1,autoxmin=0,autoymin=-1.5,autoymax=1.5
   done
   ```

   Run this a few times to make sure you are locked on a periodic. Then Run AUTO using the above parameters. You should see that the periodic solution loses stability at a point which AUTO marks as `PD` – this means that there is a period-doubling instability. Grab this point and Click on `Run` choosing `Period doubling` from the menu choices. Be ready to click on the `ABORT` key since it will circle around the period-2 orbit.

2. Find a problem in a research paper that involves periodic forcing and analyze the bifurcation behavior. Or, see what kind of phenomena you can find by adding a term like
   $$c \cos \omega t$$
   to the Morris-Lecar equations.

7.5 Importing the diagram into *XPPAUT*

In this section, I will describe a nice geometric idea that underlies the complex firing patterns of some neurons. Many neurons exhibit what is known as bursting behavior which consists of bouts of rapid spiking followed by period of relative quiescence. A typical voltage trace is shown in figure 7.9. The model used to produce this is the Morris-Lecar model (previously described) but such that the applied current satisfies a differential equation depending on the potential:

$$\begin{aligned}
C\frac{dv}{dt} &= I - g_L(V - V_L) - g_{Ca}m_\infty(v)(V - V_{Ca}) - g_K w(V - V_K) \\
\frac{dw}{dt} &= \phi\frac{w_\infty(v) - w}{\tau_w(v)} \\
\frac{dI}{dt} &= \epsilon(v_0 - v)
\end{aligned}$$

with parameters given below in the associated ODE file:

```
# mlsqr.ode
dv/dt = ( I - gca*minf(V)*(V-Vca)-gk*w*(V-VK)-gl*(V-Vl))/c
dw/dt = phi*(winf(V)-w)/tauw(V)
dI/dt = eps*(v0-v)
v(0)=-41.84
w(0)=0.002
minf(v)=.5*(1+tanh((v-v1)/v2))
winf(v)=.5*(1+tanh((v-v3)/v4))
tauw(v)=1/cosh((v-v3)/(2*v4))
param eps=.001,v0=-26,vk=-84,vl=-60,vca=120
param gk=8,gl=2,c=20
param v1=-1.2,v2=18
param v3=12,v4=17.4,phi=.23,gca=4
@ dt=1,total=4000,meth=cvode
@ xp=I,yp=v,xlo=25,xhi=45,ylo=-60,yhi=20
done
```

This is set up in the (I, v)-plane to illustrate how the current, I increases and then decreases as the voltage either moves through an apparent steady state or oscillates. Plot I against t to see that the oscillations are almost nonexistent for I. This is because ϵ is small and so the I equation is essentially satisfying:

$$\frac{dI}{dt} = \epsilon(v_0 - < v >)$$

where $< v >$ is the average of $v(t)$.

Since I is slowly varying, this suggests treating I as a parameter and studying the behavior of the (v, w) system as I varies. The file, `ml2dhom.ode` will do the trick:

```
# ml2dhom.ode
dv/dt = ( I - gca*minf(V)*(V-Vca)-gk*w*(V-VK)-gl*(V-Vl))/c
dw/dt = phi*(winf(V)-w)/tauw(V)
v(0)=-41.84
w(0)=0.002
minf(v)=.5*(1+tanh((v-v1)/v2))
winf(v)=.5*(1+tanh((v-v3)/v4))
tauw(v)=1/cosh((v-v3)/(2*v4))
param i=30,vk=-84,vl=-60,vca=120
param gk=8,gl=2,c=20
param v1=-1.2,v2=18
param v3=12,v4=17.4,phi=.23,gca=4
@ total=150,dt=.25,xlo=-60,xhi=60,ylo=-.125,yhi=.6,xp=v,yp=w
@ dsmax=2,parmin=-30,parmax=60
@ autoxmin=-10,autoxmax=60,autoymin=-60,autoymax=40
done
```

I have set this up specifically for AUTO so that it is ready to analyze the system. (The virtues of hindsight.) The last two lines of options set up the AUTO window and the numerical continuation. Run *XPPAUT* and integrate this a few times using the `Initialconds Go` and then the `Initialconds Last` commands. Then click on `File Auto` and click on `Run Steady state` in the **AUTO window** . You should see a cubic-like curve of steady states. Click on `Grab` and follow the branch until you find the Hopf bifurcation, labeled `HB` at the bottom of the **AUTO window** . Click `Enter` to grab this point. Now, click on `Run Periodic` to track the branch of periodic solutions emerging from the Hopf bifurcation. This branch terminates on the middle branch at a homoclinic orbit. The picture will be similar (but not identical) to Figure 7.4. Now for the cool part. We will save this diagram and then import it into the bursting file so that the (I, v) dynamics can be plotted superimposed on the bifurcation diagram. In the **AUTO window** , click on `File Write pts` and pick a file name such as `mlhom.dat` to save the points of the diagram. Exit *XPPAUT* . Run *XPPAUT* with the file `mlsqr.ode` which is the previously viewed bursting equation. Integrate the equations to get something like the top right figure 7.9. Now click on `Graphics Freeze Bif. Diag.` and choose the previously saved diagram, e.g., `mlhom.dat`. Voila! You will see the nice burst superimposed on the bifurcation diagram. This explains the mechanism for bursting. When the potential, v is below v_0 the current I increases causing the voltage to follow the lower stable fixed point. This continues until the "knee" is reached and the potential jumps up to the branch of periodic solutions. The average voltage is greater than v_0 so that I begins to decrease until it hits the homoclinic point. The potential then drops to the stable rest state where it repeats the cycle.

7.5.1 Exercise

Repeat the above trick with the canonical elliptic burster:

$$\begin{aligned}
u' &= u(\lambda + R - R^2) - v + c \\
v' &= v(\lambda + R - R^2) + u + c \\
\lambda' &= \epsilon(R_0 - R) \\
R &= u^2 + v^2
\end{aligned}$$

Try $\epsilon = .01, R_0 = 0.3, c = 0.01$ and integrate for a total of 1000 with `dt=0.25` using `qualrk` as the method. Plot the (λ, u)-projection in a window $-0.4 < x < 0.4$ and $-1.2 < y < 1.2$. Next write an ODE file for the (u, v) system with λ as a parameter. Start with $\lambda = -.4$ and find a steady state. Then track the fixed point as λ increases using AUTO. (Hint: set `Dsmax=0.1` and `Par Min=-0.5` in the AUTO numerics dialog.) Find the Hopf bifurcation and obtain the periodic solution. Write the points to a file. Then superimpose them on the elliptic burster (λ, U)-plane.

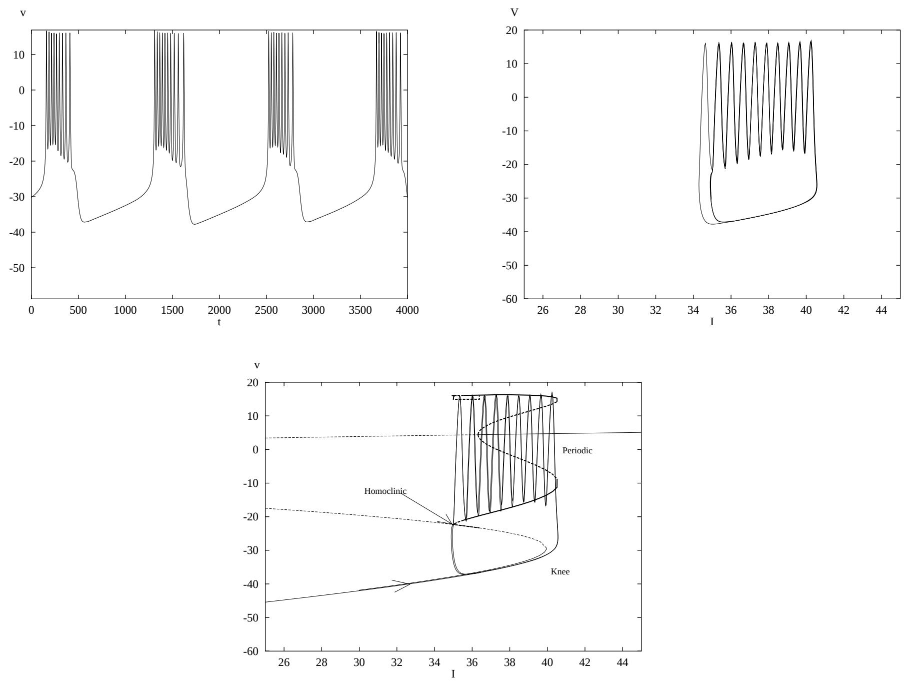
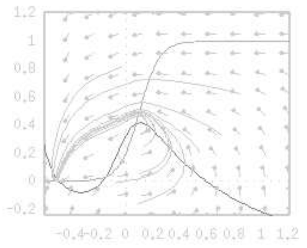


Figure 7.9. *Bursting in a modified membrane model. Top left: the voltage as a function of time during a burst; top right: the slow parameter, I and the voltage; bottom: superimposition of the bifurcation diagram in the (I, v)-plane.*



Chapter 8

Animation

8.1 Introduction

XPPAUT has a built in animator which allows you to create cartoons of the systems that you are simulating. The animation file that you create is separate from the ODE file and for a given ODE, you can load in many different animation files, experiment with them and change them without having to exit *XPPAUT* . The idea behind the animation files (which have an extension `.ani`) is to draw simple geometric objects whose coordinates and colors depend on the dynamic variables that you have integrated. The cartoon is made by redrawing the picture at each time step after changing the coordinates appropriately. Animations can be very simple one-liners or complex pictures – although, perhaps not complex enough to give Disney any competition.

In this chapter, I will take you through the basics of animation, show how it can be used for classroom demonstrations, research problems and then present a bunch of my own which are based on toys and amusement park rides.

8.2 Some first examples.

8.2.1 The pendulum

A good first example is the nonlinear pendulum which we looked at earlier:

$$ml^2\ddot{\theta} = -mgl\sin\theta.$$

The ODE file is called `pendulum.ode`. Run this with an initial value of $\theta(0) = 3$ and $\dot{\theta}(0) \equiv v(0) = 0$. You should see a nice big oscillation. Let's make some physical sense out this. Recall that θ is the angle between the end of the pendulum and the $y-$axis. Thus, the coordinates of the bob of the pendulum are

$$(x, y) = (l\sin\theta, l(1-\cos\theta)).$$

To depict the pendulum, we should draw the rod for the pendulum as well as the bob. Perhaps, we should also draw a stand for the pendulum as well, say, something like Figure 8.1. Let's do this in steps. First, we will draw the rod and then gradually build on this basic picture. Here is an animation file for the rod:

```
# pendulum.ani - goes with pendulum.ode
# animation for the pendulum
line .5;.5;.5+.3*sin(theta);.5-.3*cos(theta);$BLUE;3
done
```

The first two lines are comments and the last line tells the animator that the file is done. Thus, there is only one real line of code. Before discussing this line, I want to set out the coordinate frame. The animation window is the unit square by default so that (0,0) is at the lower left and (1,1) the upper right. Thus, the coordinate (0.5,0.5) is in the middle. This will be our pivot point for the pendulum. We will scale the length of the pendulum so that it is 3/10 of the length of the frame. This is more a matter of aesthetics; however, the animation looks pretty ugly if the drawings go out of the square since there is no clipping done. All coordinates and arguments for the drawing commands are separated by semicolons. The `line x1;y1;x2;y2` command draws a line from `(x1,y1)` to `(x2,y2)`. Thus the command tells the animator to draw a line from $(0.5, 0.5)$ to $(0.5 + 0.3\sin\theta, 0.5 - 0.3\cos\theta)$ at each time step using the value of the variable θ to compute the coordinates. The last two arguments of the `line` command tell the animator to make the rod blue and draw it with a line that is three pixels wide. In the **Main Window** , click on `View axes` `Toon` and a new window will pop up (see Fig 8.1). It has as its title some random cartoon that I like. In the **animation window** click on the `File` button. Choose the animation file `pendulum.ani` assuming you have either downloaded or typed it in and saved it. If you made errors, an error message will pop up. Otherwise, it is probably OK. In the **animation window** click on the `Go` button and it should start to swing (sort of like England?). Click on `Pause` to stop it and `Reset` to rewind it to the beginning. The left and right buttons advance by a single frame at a time. The `Skip` button lets you change the frame rate so, e.g., only one out of every 3 frames are drawn. `MPEG` allows you to create mpeg movies (with an additional program called `mpeg_encode`) or animated gifs. I describe how to do this later. Finally, there is a little checkbox in the upper right corner. If you click on this, then the animation is run simultaneously while you solve the ODE. This slows the integration of the equations down dramatically as drawing the animation is quite CPU intensive. Check the box and then change parameters or initial data and then integrate. You can watch the animation on the fly.

Change the initial conditions and re-integrate the equations. Click on `Reset` `Go` in the **animation window** and see how different it looks. Choose `theta=3,v=.5` and watch the pendulum. It windmills. Now, lets add the bob and the frame:

```
# pendulum.ani - goes with pendulum.ode
# animation for the pendulum
#
# the frame
line .5;.1;.9;.1;$GREEN;2
```

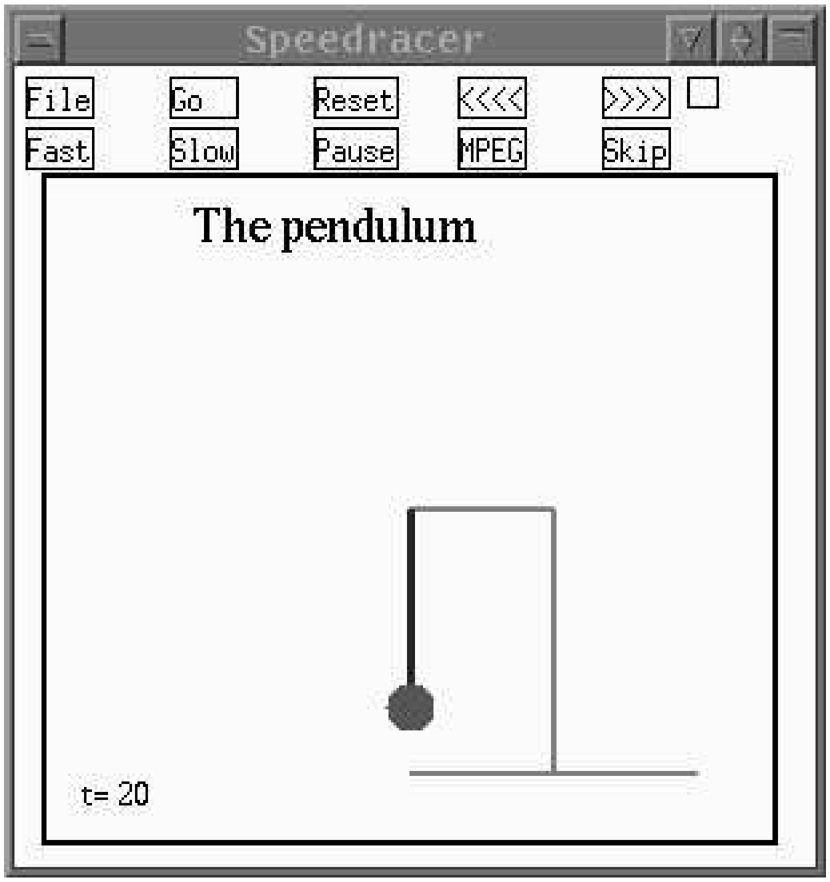


Figure 8.1. *The animation window*

```
line .5;.5;.7;.5;$GREEN;2
line .7;.5;.7;.1;$GREEN;2
#
# the pendulum
line .5;.5;.5+.3*sin(theta);.5-.3*cos(theta);$BLUE;3
fcircle .5+.3*sin(theta);.5-.3*cos(theta);.04;$RED
done
```

For the green frame, we have drawn a series of three line segments which are two pixels wide. The new command starts `fcircle x;y;r` and draws a filled circle centered at `(x,y)` with radius `r`. The last option is just the color. Thus, we draw a bob that is a filled red circle centered at the end of the rod and with a radius of 0.04.

Edit your old `pendulum.ani` file and change it as above. In the **animation window** , click on `File` and `Ok` since this file should be the default now as you already loaded it. Click on `Go` and you will see the full animation.

Note that we really don't have to draw the frame every time as it never changes; it should be drawn only once. The animator has a control for that, `PERM` that says that every drawing command that follows only has to be calculated at the beginning and doesn't change thereafter. The opposite command is `TRANS` which tells the animator to recompute every time. So, we could improve this slightly by adding `PERM` and `TRANS` to the file. We will also add a nice title in black:

```
# pendulum.ani - goes with pendulum.ode
# animation for the pendulum
#
PERM
# the frame
settext 3;roman;$BLACK
text .2;.9;The pendulum
line .5;.1;.9;.1;$GREEN;2
line .5;.5;.7;.5;$GREEN;2
line .7;.5;.7;.1;$GREEN;2
#
TRANS
# the pendulum
line .5;.5;.5+.3*sin(theta);.5-.3*cos(theta);$BLUE;3
fcircle .5+.3*sin(theta);.5-.3*cos(theta);.04;$RED
done
```

Note that the `settext` command takes three arguments; the text size (0-5), the font (roman, symbol) and a color. The default is black roman with size 0 which is really tiny. **NOTE** I have set the animator back to transient mode before the commands for actually drawing the pendulum. If you don't do this, only the first frame will be drawn. Load this animation into the animator and make it go!

We will introduce one more bell and whistle. In small text, we will type the current time of the simulation. Here is the final version

```
# pendulum.ani - goes with pendulum.ode
# animation for the pendulum
#
PERM
# the frame
settext 3;roman;$BLACK
text .2;.9;The pendulum
line .5;.1;.9;.1;$GREEN;2
line .5;.5;.7;.5;$GREEN;2
line .7;.5;.7;.1;$GREEN;2
settext 1;roman;$BLACK
#
TRANS
# the pendulum
```

```
vtext .05;.05;t= ;t
line .5;.5;.5+.3*sin(theta);.5-.3*cos(theta);$BLUE;3
fcircle .5+.3*sin(theta);.5-.3*cos(theta);.04;$RED
done
```

I have first set the text back to black and set the size to 1. The command `vtext` takes four arguments: the position on the screen, a text string, and a formula which will be evaluated at each step. Thus, we will put the string `t=` on the screen followed by the value of t.

Finally, let's make a permanent version of your animation. There are two ways to do this, an easy and a hard way. The hard way is to first produce a sequence of `ppm` files and then combine them with `mpeg_encode`. This technique produces smaller files but requires an enormous amount of disk space. The manual that comes with *XPPAUT* explains how to do this in detail. Instead, we will use the easy way; make an animated GIF file. Animated GIF is a standard used to produce those annoying animations that populate many web pages. The files can be "played" using any browser. Suppose you have an animation you are happy with. Then click on the `MPEG` button in the **animation window** . In the resulting dialog box, change the 0 to a 1 in the entry labeled `AniGif`. Close the dialog box. Now click on `Skip` and change the `Increment` from 1 to 2 which plots every other frame. Click on `Reset` to reset the animator to the beginning of the data file. Then click on `Go` in the **animation window** and the animator will start up again. It may pause on occasion as it writes to the disk. After a while, 200 frames will be written into a file called `anim.gif`. The file is always called this so if you want to have multiple animations, make sure you rename it after you are done. The file `anim.gif` is about 225000 bytes which is pretty small for a 200 frame animation. Try playing it by loading it into your browser or using software like `animate` and `xanim` on Linux and `QuickTime Player` and `RealPlayer` on Windows. (Note that the execrable `Media Player` doesn't support animated GIFs). Now you can fill your web page with fancy dynamical systems animations and further clog the bandwidth!

8.2.2 Important notes

The animator is aware of all of the state variables in your system but is not aware of any of the auxiliary quantities. However, it is aware of fixed variables. Thus, if you want to pass on a complicated quantity to the animator, it is best to define it in the ODE file rather than in the animation file. Furthermore, you should define it as a **fixed** variable rather than an **auxiliary** variable. For example in the pendulum ODE file we could add the lines

```
xb=.5+.3*sin(theta)
yb=.5-.3*cos(theta)
```

to define the endpoints of the bob. Then the animation file would be much simpler:

```
line .5;.5;xb;yb;$BLUE;3
fcircle xb;yb;.04;$RED
```

8.2.3 A spatial problem revisited

Recall in chapter 6 that we looked at a spatially distributed neural network with a delay:

$$u_j' = -u_j + f(c_e \sum_{l=-m}^{m} W(l)u(j+l) - c_i u_j(t-\tau)).$$

The ODE file was called `nnetdelay.ode`. Run this again. Here, we will create an animation file which plots a series of small filled circles at each spatial point along the x-axis and whose height is a scaled version of the state variable u. Here is the file `nnetdelay.ani`

```
# use with nnetdelay.ode
# a one liner!
fcircle .05+.9*[0..99]/100;.05+.8*u[j];.01
done
```

Recall that the `[0..99]` statement is expanded by *XPPAUT* into 100 lines and each `[j]` is substituted by the corresponding integer. Thus, this animation code represents a long file:

```
fcircle .05+.9*0/100;.05+.8*u0;.01
fcircle .05+.9*1/100;.05+.8*u1;.01
...
fcircle .05+.9*99/100;.05+.8*u99;.01
```

I have scaled everything so that it lies in the window. Load this animation file and look at the result.

In fact, this is not an optimal way to view the time evolution. It would be much better to draw lines between the points. This is easy as well with the one-line file `nnetdelay2.ani`

```
# use with nnetdelay.ode
# a one liner!
line .05+.9*[0..98]/100;.05+.8*u[j];.05+.9*[j+1]/100;.05+.8*u[j+1];$BLACK;2
done
```

Note that I only go from 0 to 98 rather than 0 to 99. This draws a black line with thickness 2 between successive `(x,y)` values with `x` the spatial grid point and `y` the magnitude of u. Load this animation file and try it.

8.2.4 A glider and a fancy glider

The little balsa wood glider that is popular with everyone is readily modeled as a pair of differential equations. I will not go through a complete derivation of the equations. Consider a glider that is moving in the (x, y) plane (so that it can only tilt up and down and cannot veer out of the plane.) The dimensionless equations are for the speed, v, (a scalar quantity which is the magnitude of the velocity) and the attack angle, θ, the angle the nose makes with the horizontal. These equations are due to Lanchester (1908) and the model is taken from the book by West, et. al.(1997). The equations are

$$\frac{dv}{dt} = -\sin\theta - Dv^2 \qquad v\frac{d\theta}{dt} = v^2 - \cos\theta.$$

The $-Dv^2$ is the viscous drag and works against the speed. The term v^2 is the lift. The component of gravity along the direction of the plane is $-mg\sin\theta$ and since the equations are dimensionless, we set the coefficients to 1. The term $vd\theta/dt$ is centripetal force and the $\cos\theta$ term is the component of gravity in the angular direction. These equations only give the speed and the attack angle. We also need to track the position of the plane which is found by integrating the velocity, $\vec{v} = (v\cos\theta, v\sin\theta)$:

$$\frac{dx}{dt} = v\cos\theta \qquad \frac{dy}{dt} = v\sin\theta.$$

Simple glider

We can go ahead and solve these equations numerically using *XPPAUT* and then create an animation file. For our first version, we will plot a thick line to represent the glider. The position on the screen is given by a scaled version of (x, y) and the angle of the line is θ. Here is the ODE file, `glider.ode`

```
# glider
v'=-sin(th)-d*v^2
th'=(v^2-cos(th))/v
x'=v*cos(th)*scale
y'=v*sin(th)*scale
par d=.25,scale=.1,lg=.04
# these initial data correspond to a nice loop
init th=0.25,v=3
init x=.1,y=.2
# some stuff for the animation
# glider is just a thick line
# 2*lg=length glider
xl=mod(x,1)-lg*cos(th)
yl=mod(y,1)-lg*sin(th)
xr=mod(x,1)+lg*cos(th)
yr=mod(y,1)+lg*sin(th)
# some controls
@ total=30,dt=.025,bound=1000000
done
```

The first part of the file contains all the equations we discussed above. We have scaled the (x, y) coordinates so that they don't get too big. The last part of the ODE file contains information to pass on to the animator. I first mod out the coordinates by 1 so that when the glider passes out one boundary of the animator, it comes back in the other – the world is a torus. These new coordinates form the center of the glider. Then I construct the left and right endpoints of the "glider". The net size of the glider is `2*lg`. Run *XPPAUT* with the file `glider.ode` and then run the animation. Change the drag, `d` making it lower and see if you can execute two loops.

A fancy glider

Now, we will build a fancier glider that looks more like a plane. (Well maybe only a little more.) This time, I will use the lookup tables in *XPPAUT* for the coordinates of the fancy glider. The tables are called `xglid.tab` and `yglid.tab`. Here are the 14 values of the two coordinates

```
xglid.tab: 0 2.5 2 .5 -1 -1.75 -.5
           -2 -2.5 -2.5 -1 -2 -1.25 0
yglid.tab: 0 0 1 1 2 2 1
           1 2 0 0 -1 -1 0
```

Here is the ODE file, `fglider.ode`

```
# fancy glider
tab xg xglid.tab
tab yg yglid.tab
v'=-sin(th)-d*v^2
th'=(v^2-cos(th))/v
x'=v*cos(th)*scale
y'=v*sin(th)*scale
par d=0.25,scale=2,lg=.04
# these initial data correspond to a nice loop
init th=.2,v=8
init x=4,y=0
# these are points for the fancy glider
xc=x
yc=y
c=cos(th)
s=sin(th)
xx[1..14]=(xg([j])*c-yg([j])*s+xc)*lg
yy[1..14]=(xg([j])*s+yg([j])*c+yc)*lg+.5
@ total=30,dt=.025,bound=1000000
done
```

The structure is similar to the plane glider but the scaling is done after making the glider. The picture for the glider is centered at the origin, so the coordinates must be rotated by the angle θ and then translated by the center of mass of the glider. Thus, I define 14 points for the x and y coordinates of the glider which rotate the little picture and translated it by the center position. Finally, I scale the whole thing and lift up the y coordinate a bit. I don't bother to mod out the coordinates so that it can fly off into the sunset at its leisure. The ODE file does all the work and the animation file is but one line, `fglider.ani`:

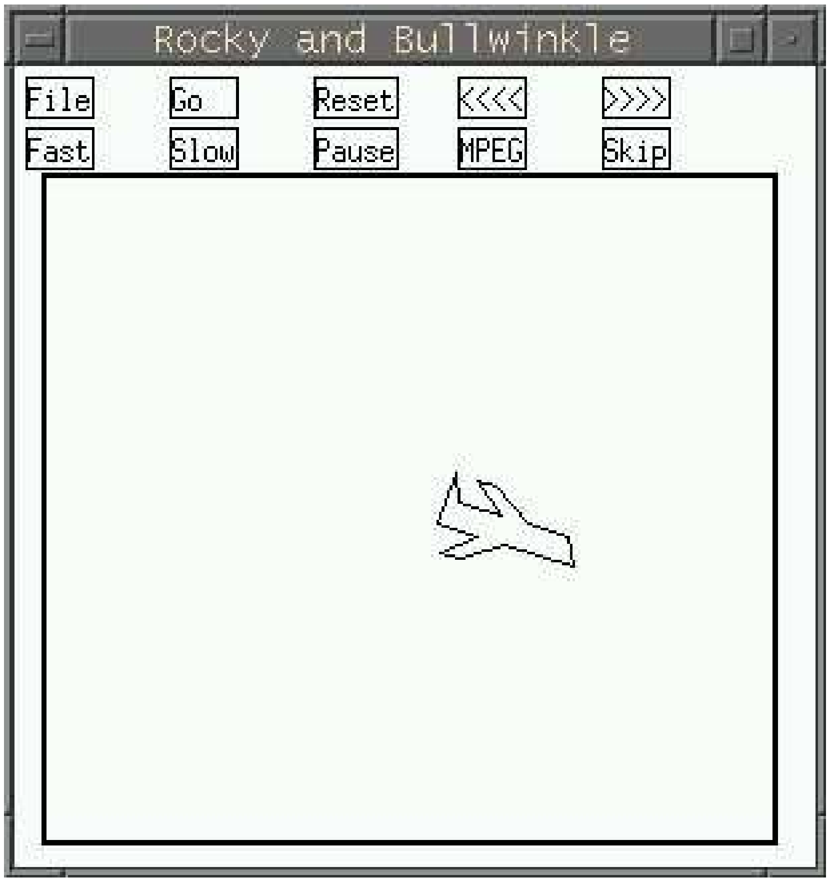


Figure 8.2. *The fancy glider!*

```
# fancy animation for glider
line xx[1..13];yy[j];xx[j+1];yy[j+1]
end
```

I connect the lines for the glider – it was designed as a closed curve. Try these two files.

8.2.5 Coupled oscillators

Recall that we looked at the Morris-Lecar model in a number of prior chapters. Suppose that we take a cluster of such oscillators and couple them together with diffusion through the average voltage of all of them:

$$V_j' = I_{tot}(V_j, w_j) + d(\frac{1}{N}\sum_k V_k - V_j)$$

where I_{tot} is the local ionic current of each cell. We can ask whether they will lock to a periodic orbit and if so whether it will be synchronous or whether other more complex solutions are possible. The file `ml10.ode` implements a coupled array of ten such neural oscillators:

```
# ml10.ode
# morris-lecar; dimensionless
vtot=.1*sum(0,9)of(shift(v0,i'))
wtot=.1*sum(0,9)of(shift(w0,i'))
itot(v,w)=I+gl*(el-v)+gk*w*(ek-v)+gca*minf(v)*(eca-v)
g(v,w)=(winf(v)-w)*lamw(v)
v[0..9]'=itot(v[j],w[j])+d*(vtot-v[j])
w[0..9]'=g(v[j],w[j])
par d=.05
par I=.095,phi=1
par ek=-.7,eca=1,el=-.5
par gl=.5,gk=2,gca=1
par va=-.01,vb=0.15,vc=0.1,vd=0.145
minf(v)=.5*(1+tanh((v-va)/vb))
winf(v)=.5*(1+tanh((v-vc)/vd))
lamw(v)= phi*cosh((v-vc)/(2*vd))
@ total=200,nout=4
@ xlo=-.6,xhi=.5,ylo=-.25,yhi=.75
@ xp=v0,yp=w0
done
```

Notes. The term `sum(0,9)of(shift(v0,i'))` sums up all the voltages similarly for the recovery variables – we will use the latter in the animation. I have set the output to every fourth point to make the animation go faster.

Run *XPPAUT* and use the `Initialconds Formula` command setting the variables `v[0..9]` to `-.5+.5*ran(1)` which assigns them random voltages between -0.5 and 0.5. Now fire up the animator and load the animation file `ml10.ani`

```
# ml10.ani
# use with ml10.ode
fcircle v[0..9]+.5;w[j]+.1;.02;[j+1]/10
fcircle vtot+.5;wtot+.1;.03;$BLACK
end
```

This file draws a filled circle in the (v, w) plane for each cell and colors it by its index so that it is easy to see each individual. Also, the centroid (the average voltage and recovery variable) are plotted as a bigger black circle. Run the animation and watch the cells spread out to almost uniformly fill in the limit cycle; the centroid stays near the inside and doesn't change much. Now change `phi` from 1 to 0.5 and re-integrate the equations (`Initialconds Go`). Now look at the animation. Use the techniques of weakly coupled oscillators(see Chapter 9) to explain the difference between `phi=1` and `phi=0.5`.

You can make this file even better by superimposing the nullclines onto the animation. It is best to do all the calculations needed in the ODE file rather than in the animation file, so that we add the following lines of code to the ODE file:

```
##   nullcline stuff for animation
# define the nullclines
wnull(v)=winf(v)
vnull(v)=(I+gl*(el-v)+gca*minf(v)*(eca-v))/(gk*(v-ek))
# now crudely draw them for the animation
vv[0..20]=-.6+1.1*[j]/20
vn[0..20]=vnull(vv[j])+.25
wn[0..20]=wnull(vv[j])+.25
done
```

I have drawn 20 line segments for each nullcline and also scaled them identically to the phaseplane drawn in the ODE file. The points `vv[j], vn[j]` are the coordinates for the v-nullcline while `vv[j],wn[j]` are those for the w-nullcline. With these added lines in the ODE file, the animation file now has the form:

```
# ml10x.ani
# use with ml10x.ode
PERM
line [0..19]/20;vn[j];[j+1]/20;vn[j+1];$BLUE;3
line [0..19]/20;wn[j];[j+1]/20;wn[j+1];$RED;3
TRANSIENT
fcircle (v[0..9]+.6)/1.1;w[j]+.25;.02;[j+1]/10
fcircle (vtot+.6)/1.1;wtot+.25;.03;$BLACK
end
```

The nullclines are drawn along with the values of the variables.

8.3 My favorites.

In this section, I will describe a number of my favorite toys and mechanical objects. I will start with some variants of the pendulum. Then I will create a roller-coaster model and a model of a great ride, the **Zipper**. I will finish with the discrete analogue of the Lorenz equations following Strogatz.

8.3.1 More pendula

The plain old pendulum is rather boring. However, if you do things like periodically drive the pendulum or glue two of them together, they produce pretty complex behavior. They are also real fun to watch. The first example I want to do is the parametrically driven pendulum. As you know, the down position is asymptotically stable for a damped pendulum and the straight-up state is unstable. However, if you put the pendulum on an oscillating platform, you can stabilize the up state. This is how jugglers keep the broomstick standing up balanced on the palm of their hand – they jiggle it. The differential equations are not hard to derive. (See for example, Guckenheimer and Holmes). You write down the Lagrangian (see the next section) and from this derive the equations of motion. The result is the following

$$ml^2\ddot{\theta} = -f\dot{\theta} - mg\sin\theta + aml\omega^2 \sin\omega t \sin\theta$$

where a is the amplitude of the sinusoidal forcing and ω is the frequency. Here is the corresponding ODE file, `forcpend.ode`:

```
# parametrically forced pendulum
#  forcpend.ode
yo(t)=a*sin(w*t)
yopp(t)=-a*w*w*sin(w*t)
th'=thp
thp'=(-f*thp-m*g*sin(th))/(m*l*l)-yopp(t)*sin(th)/l
par l=2,g=9.8,m=1
par a=0.3,w=50,f=.25
init th=2
@ meth=qualrk
@ total=100,bound=10000,tol=1e-5,atol=1e-4
done
```

Start up *XPPAUT* with this file and run the integration. Here is the animation file, `forcpend.ani`

```
# forcpend.ani
# force pendulum
# use with forcpend.ode
#
# the oscillating base:
line .4;.5+.2*yo(t);.6;.5+.2*yo(t);$BLUE;2
# the pendulum
line .5;.5+.2*yo(t);.5+.2*l*sin(th);-.2*l*cos(th)+.5+.2*yo(t);$BLACK;4
fcircle .5+.2*l*sin(th);-.2*l*cos(th)+.5+.2*yo(t);.05;$RED
end
```

The base is just a horizontal blue line which oscillates up and down in the vertical direction. The pivot point of the pendulum is centered on this point and the bob end of the pendulum is determined by the angle, θ. The scaling factor 0.2 which multiplies all the dimensions is just to keep everything in the window. Load this into the animator and let it rip! Notice how the pendulum goes right up. Try this experiment: set the frequency to a low value, `w=4`, the friction, `f=.03` to a low value, and the amplitude `a=.3` and integrate the equations. The pendulum chaotically switches back and forth. Animate it.

The next example is another classical mechanics standard, the double pendulum. This is a four-dimensional system. As with all mechanical systems, you derive the differential equations by writing down the potential and kinetic energy. They are complicated and instead of the equations, I will just give you the ODE file, `doubpend.ode`:

```
 # the double pendulum
# doubpend.ode
#  th1 is the angle between the pivot and the first mass
#  th2 is the angle between the first mass and the second mass
#  mu is friction
# L1-2, m1-2 are the length and mass.
th1' = th1p
th2' = th2p
th2p' = -mu*th2p+1/(L2*(m1+m2*(sin(th2-th1))^2))*\
(-(m1+m2)*g*sin(th2)-\
(m1+m2)*L1*th1p^2*sin(th2-th1)+cos(th2-th1)*\
((m1+m2)*g*sin(th1)-\
m2*L2*th2p^2*sin(th2-th1)))
th1p' = -mu*th1p+1/(L1*(m1+m2*(sin(th2-th1))^2))*\
(-(m1+m2)*g*sin(th1)+\
m2*L2*th2p^2*sin(th2-th1)+cos(th2-th1)*(m2*g*sin(th2)+\
m2*L1*th1p^2*sin(th2-th1)))
# parameters
par L1=.1,L2=.1,m1=.028,m2=.04,g=9.8,mu=.01
init th1=0,th2=0,th2p=31,th1p=0
@ total=30,dt=.01,meth=qualrk,bound=100000
done
```

This is set up to simulate two 10 cm pendula with a masses of 28 and 40 grams respectively. I give the lower one a good spin. Time units are in seconds so the whole simulation represents a half a minute. Integrate these equations. Before we get to the animator, convince yourself that the solution is chaotic. Compute the maximal Liapunov exponent (`nUmerics stocHastic Liapunov`) and also compute the power spectrum of the angular velocity, `th1p` of the lower mass (`nUmerics stocHastic Power`). Now, here is the animation file, `doubpend.ani`

```
# animation of double pendulum
#
# here is the base
fcircle .5;.5;.025;$BLACK
# first rod
line .5;.5;.5+.2*sin(th1);.5-.2*cos(th1)
# second rod
line .5+.2*sin(th1);.5-.2*cos(th1);.5+.2*sin(th1)+.2*sin(th2);\
.5-.2*cos(th1)-.2*cos(th2)
# bobs
fcircle .5+.2*sin(th1);.5-.2*cos(th1);.04;$RED
fcircle .5+.2*sin(th1)+.2*sin(th2);.5-.2*cos(th1)-.2*cos(th2);.04;$GREEN
end
```

This should be self explanatory. Use this to look at the chaotic behavior.

There are many other related equations and I urge you to look at them.

8.3.2 A roller coaster

How does one simulate a roller coaster? Again, we appeal to classical mechanics. We will parameterize the coaster by a variable s which you can think of as an arclength. We will also restrict the coaster to a plane (although it will be allowed to loop. Let $x(s), y(s)$ define the coaster and assume that $s \in [0,1]$. Let m be the mass of the car. The kinetic energy is

$$K = \frac{m}{2}(\dot{x}^2 + \dot{y}^2) = \frac{m}{2}(x'(s)^2 + y'(s)^2)\dot{s}^2.$$

The potential energy is just

$$P = mgy(s).$$

The Lagrangian is $L = K - P$. The equations of motion are given by the Euler-Lagrange equations:

$$\frac{d}{dt}\left(\frac{\partial L}{\partial \dot{s}}\right) = \frac{\partial L}{\partial s}.$$

This leads to:

$$mR(s)\ddot{s} = -mgy'(s) - A(s)\dot{s}^2$$

where

$$R(s) = x'(s)^2 + y'(s)^2 \qquad A(s) = x'(s)x''(s) + y'(s)y''(s).$$

This is a general theory of roller coasters. Now we need to design one. I designed a very simple one with one loop using the program `xfig` which produces text output that consists of the coordinates of the points on the coaster. I then extracted from these points, two tables of the x, y values which I stuck in two tables, `xcoast.tab` and `ycoast.tab` (available on the web). Here is the ODE file for the coaster, `coaster2.ode`

```
# coaster2.ode
# this is a rather simple coaster
tab xx xcoast.tab
tab yy ycoast.tab
# numerically compute x',x'' etc
xp=(xx(s+h)-xx(s-h))/(2*h)
yp=(yy(s+h)-yy(s-h))/(2*h)
xpp=(xx(s+h)-2*xx(s)+xx(s-h))/(h*h)
ypp=(yy(s+h)-2*yy(s)+yy(s-h))/(h*h)
#
vsq=xp*xp+yp*yp
aa=(xp*xpp+yp*ypp)/vsq
# now add a few parameters to scale time and gravity etc
s'=v
v'=-r*yp/vsq-q*aa*v*v
par r=175,h=.02,q=.1
# give it a little push!
init v=1e-5
```

```
# here is the coaster
aux x=xx(s)
aux y=yy(s)
@ bound=100000,total=40
@ xp=x,yp=y,xlo=0,ylo=0,xhi=12000,yhi=4500,lt=0
done
```

I have used the table function from *XPPAUT* to load in the coordinates of the coaster. I also numerically differentiate these two functions. (**Note** that I have set a flag in the table files that tells *XPPAUT* to use a cubic interpolation rather than the default linear interpolation for evaluating the tables. This way, the derivatives are a little smoother. The flag is set in the first line of the table file as a letter preceding the number of entries of the table. Here are the first few lines of the table:

```
s35
0
1
300
600
900
```

The letter is `s` for spline – the tables `xcoast.tab` and `ycoast.tab` are available on the web). I have also set the plotting window to show the actual shape of the coaster as well as the velocity. Since the coaster was defined with more grid points along the loops, it is not really completely physically correct and things slow down along the curves more than they should.

8.3.3 The Zipper

The Zipper (Chance Rides) is a carnival ride that consists of a large ovoid track along which freely hanging cars move. In addition, the whole track moves in a circle. The ride is full of action with the chairs spinning randomly as it makes it way around the track. Figure 8.3 shows a sketch of the Zipper. The freely hanging car with riders is essentially an underdamped pendulum. If all the Zipper did was move the cars around the track, it would be a periodically forced pendulum. However, the additional rotation about the central axis makes this a quasi-periodically forced pendulum. Thus, we can expect the motion to be quite complex. Conveniently, one can find the manufacturer of this ride (www.rides.com) and get all the specifications such as the rate of rotation of the track and the dimensions of the ride (see the figure which includes the manufacturer's diagram)

The hardest part of this simulation is to create the track. I use a trick to do this. I create an integrable dynamical system of the form

$$x' = afy|y|^p \quad y' = -fx|x|^p$$

whose solutions are closed curves of various amplitudes and aspect ratios. The parameter a determines the aspect ratio, the parameter f the frequency around the

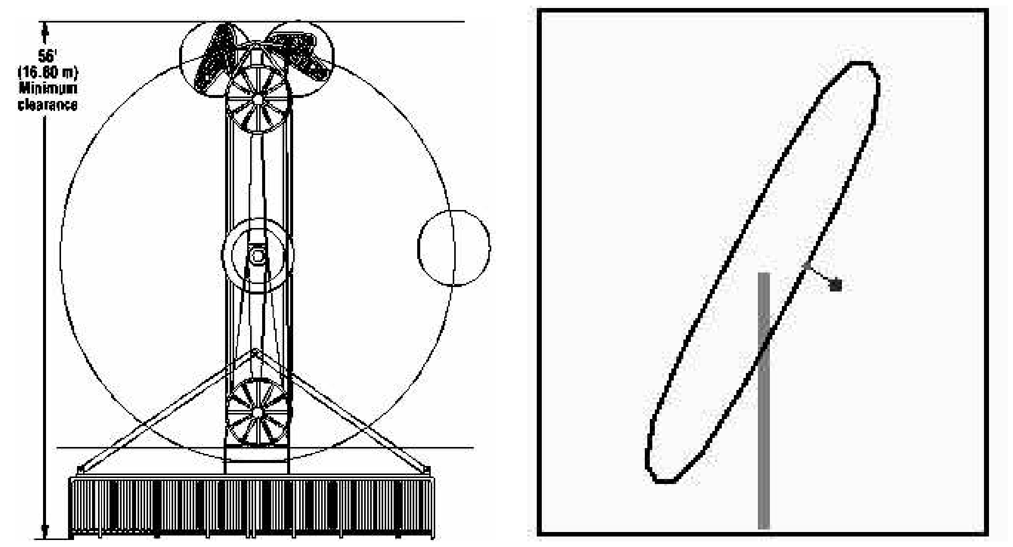


Figure 8.3. *The Zipper carnival ride. Left is the manufacturer's schematic; right shows the Zipper as rendered by XPP*

system, and the initial data determine the overall size of the track. The positions x, y of this system determine the position on the track of the car. To get the zipper to rotate, we will solve the harmonic oscillator

$$x_b' = \omega y_b \quad y_b' = -\omega x_b$$

where ω is the rotation frequency. Once we have the rotation position and the position on the track, we know the position of the pivot of the car. The position of the bob of the pendulum (corresponding to the center of mass of the riders) is then just as with the regular pendulum. We can then derive equations for the angle of the pendulum relative to the y-axis:

$$ml^2\ddot{\theta} = -(mgl\sin\theta + ml\ddot{x}_c\cos\theta + m\ddot{y}_c\sin\theta)$$

where $(\ddot{x}_c, \ddot{y}_c)$ is the second derivative of the position of the car on the track. The ODE file is a bit complicated by the fact that we need to draw the track. Here is the file `zipper.ode`:

```
# this is a model for the zipper carnival ride
# zipper.ode
#
# table of 16 points on the zipper
table zx zipx.tab
table zy zipy.tab
# here is the track RHSs
```

```
xdot=a*f*y*abs(y)^p
ydot=-f*x*abs(x)^p
# second derivatives
xddot=a*f*(p+1)*ydot*abs(y)^p
yddot=-f*(p+1)*xdot*abs(x)^p
# odes for the track
x'=xdot
y'=ydot
# omega and f are chosen so that they agree with the real
# machine - one is the angular frequency and the other the track
# g is gravity
# mu is friction
# l (feet) is the effective length of the cars
# a is aspect ratio for the track so that it is about 8 feet wide
# s is a scaling factor for the animation
par p=1,a=.005,l=4.5,f=.6,g=32,mu=.01
par s=60,q=1
# the initial condition here scales the length of the track 2*28=56 feet
init x=0,y=28
par omega=.8
# the endpoints of the cars
#
# now the angular variable for the zipper
xbdot=yb*omega
ybdot=-xb*omega
xbddot=-omega*omega*xb
ybddot=-omega*omega*yb
# simple harmonic oscillator for the rotation of the zipper track
xb'=xbdot
yb'=ybdot
init xb=1
#
# now put into a rotating frame
xc=x*xb-y*yb
yc=x*yb+y*xb
xcdot=xdot*xb+x*xbdot-ydot*yb-y*ybdot
xcddot=xddot*xb+2*xdot*xbdot+x*xbddot-yddot*yb-2*ydot*ybdot-y*ybddot
ycdot=xdot*yb+x*ybdot+ydot*xb+y*xbdot
ycddot=xddot*yb+2*xdot*ybdot+x*ybddot+yddot*xb+2*ydot*xbdot+y*xbddot
# this is car's dynamics
# now we add the pendulum with friction to the whole thing
th'=v
v'=-(g*sin(th)+mu*v+xcddot*cos(th)+ycddot*sin(th))/l
# position of people in the car
# scaled a bit for the animation
xp=xc+q*l*sin(th)
```

```
yp=yc-q*l*cos(th)
# position of a point on the track
aux xcar=xc
aux ycar=yc
# position of the people in a car
aux xpeople=xp
aux ypeople=yp
#
# now for the animation, we will make the track
xt[0..17]=zx([j])*xb-zy([j])*yb
yt[0..17]=zx([j])*yb+zy([j])*xb
@ xp=x,yp=y,xlo=-30,ylo=-30,xhi=30,yhi=30
@ dt=.05,total=240,bound=10000
done
```

It looks more complicated than it really is; most of the file is stuff to compute the second derivative of the forcing function. The animation file is real simple as we have already done all the work in the ODE file. Here is `zipper.ode`

```
# animation file for the zipper
perm
line .5;0;.5;.5;$GREEN;5
trans
# here is the track
line .5+xt[0..15]/s;.5+yt[j]/s;.5+xt[j+1]/s;.5+yt[j+1]/s;$BLACK;2
fcircle .5+xc/s;.5+yc/s;.015;$RED
fcircle .5+xp/s;.5+yp/s;.02;$BLUE
line .5+xc/s;.5+yc/s;.5+xp/s;.5+yp/s
end
```

The first line draws the "stand" for the zipper. The next line is the track. A red circle marks the pivot point on the track and a blue one the "car". A line joins them.

This company has many interesting rides which await analysis! For example, try to model their ride called *CHAOS* and see if it is appropriately named. It consists of a rotating tilted wheel with freely swinging cars. Thus, it is essentially a periodically forced pendulum. Who ever thought that Melnikov and transverse homoclinics could be so fun!

8.3.4 The Lorenz equations

Strogatz (1990) derives the famous Lorenz equations by looking at the leaky waterwheel. Consider an array of small cups placed around the rim of a bicycle wheel. Suppose that each cup has a small hole in it. There is a source of water at 12 o'clock that fills the cups. As they fill, the waterwheel becomes unbalanced and rotates allowing another cup to take its place. In the meantime, the previously filled cup drains slowly because of the hole. Strogatz shows that in the limit as the number of cups becomes a continuum, the resulting dynamics are completely described by the Lorenz equations. The angular velocity is the x coordinate and the y, z coordinates are the cosine and sine components of the mass of the system (which is distributed periodically on the wheel). So, with this as a motivation, I present a waterwheel with 10 cups. Water flows in at a rate `flow`. Each cup has a mass, so that like everything else we have looked at, this waterwheel is essentially 10 rigid pendula aligned in a circle. There are 10+2 differential equations: the first 10 are for the amount of water in each cup and the other two are the dynamics of the wheel – rotation angle and angular velocity. The ODE file, `waterwheel.ode` is :

```
# the waterwheel ala Lorenz but discrete
# there are 10 cups and each leaks
# here we figure out which cup is under the spigot
ff(u)=heav(cos(u)-cos(pi/n))
# flow into cup i. i=0..9
flow[0..9]=flow*ff(theta-2*pi*[j]/n)
# input-output for each cup
cp[0..9]=flow[j]-mu*c[j]
# mass of the cups
m[0..9]=c[j]+mc/n
# ODE for the cup
c[0..9]'=cp[j]
# total change in mass
mdot=sum(0,9)of(shift(cp0,i'))
# total mass
m=sum(0,9)of(shift(c0,i'))+mc
# the waterwheel equations using 10 pendulums with mass of each cup
theta'=thetap
thetap'=(-nu*thetap-l*l*mdot*thetap+l*sum(0,9)of(shift(m0,i')\
*sin(theta-2*pi*i'/n)))/(m*l*l)
# main parameter to change is flow - the spigot rate
par flow=.5,mu=.1,n=10,l=.15,mc=2,nu=.1
init theta=.05
### some stuff for animation
x[0..9]=.3*sin(theta-2*pi*[j]/n)+.4
y[0..9]=.3*cos(theta-2*pi*[j]/n)+.4
yc[0..9]=.3*cos(theta-2*pi*[j]/n)+.4+.1*c[j]/2
@ total=200,dt=.05,meth=cvode,tol=1e-5,atol=1e-4
@ yp=thetap,xhi=200,ylo=-2.5,yhi=2.5
```

Notes. I need to get the total change in mass as well as the total mass; for this I use the `sum` command. I have defined a bunch of variables `x,y,yc` which are used in the animation file. They represent the spokes of the waterwheel and the height to the water in each cup. Run this ODE and then load the animation file `waterwheel.ani`:

```
# waterwheel.ani
# waterwheel animation file
line .4;.4;x[0..9];y[j];[j]/10
line x[0..9];y[j];x[j];yc[j];$BLUE;4
done
```

The first part of the file draws the spokes of the waterwheel. The second part draws little thick blue lines to represent the height of the water in the cups.

8.4 The animator scripting language.

Animation files consist of nothing more than a series of commands to the animator. These commands are usually drawing commands, "state" commands, or style commands. Without further ado, here are all the allowable commands:

- **dimension** xlo;ylo;xhi;yho
- **speed** delay
- **transient**
- **permanent**
- **line** x1;y1;x2;y2;color;thickness
- **rline** x1;y1;color;thickness
- **rect** x1;y1;x2;y2;color;thickness
- **frect** x1;y1;x2;y2;color
- **circ** x1;y1;rad;color;thickness
- **fcirc** x1;y1;rad;color
- **ellip** x1;y1;rx;ry;color;thickness
- **fellip** x1;y1;rx;ry;color
- **comet** x1;y1;type;n;color
- **text** x1;y1;s
- **vtext** x1;y1;s;z
- **settext** size;font;color
- **end**

That's it! In all commands except **settext**, the **color** and **thickness** entries are optional. All commands can be abbreviated to their first three letters and case is ignored. At startup the dimension of the animation window in user coordinates is (0,0) at the bottom left and (1,1) at the top right. Thus the point (0.5,0.5) is the center no matter what the actual size of the window on the screen.

The **transient** and **permanent** declarations tell the animator whether the coordinates have to be evaluated at every time or if they are fixed for all time. The default when the file is loaded is **transient.** Thus, these are just toggles between the two different types of objects.

Color is described by either a floating point number between 0 and 1 with 0 corresponding to the lowest value in your color map (blue in the default) and

1 to the highest (red in the default). When described as a floating point number, it can be a formula that depends on the variables. In all the commands, the color is optional *except* **settext.** The other way of describing color is to use names which all start with the $ symbol. The names are: **$WHITE, $RED, $REDORANGE, $ORANGE, $YELLOWORANGE, $YELLOW, $YELLOWGREEN, $GREEN, $BLUEGREEN, $BLUE,$PURPLE, $BLACK**.

The number following the **speed** declaration must be a nonnegative integer. It tells the animator how many milliseconds to wait between pictures. Thus it is really "anti-speed."

The **dimension** command requires 4 numbers following it. They are the coordinates of the lower left corner and the upper right. The defaults are (0,0) and (1,1).

The **settext** command tells the animator what size and color to make the next text output. The size must be an integer, **{ 0,1,2,3,4 }** with 0 the smallest and 4 the biggest. The font is either **roman** or **symbol.** The color must be a named color and not one that is evaluated.

The remaining ten commands all put something on the screen.

- **line x1;y1;x2;y2;color;thickness** draws a line from **(x1,y1)** to **(x2,y2)** in user coordinates. These four numbers can be any expression that involves variables and fixed variables from your simulation. They are evaluated at each time step (unless the line is **permanent**) and this is scaled to be drawn in the window. The **color** is optional and can either be a named color or an expression that is to be evaluated. The **thickness** is also optional but if you want to include this, you must include the **color** as well. **thickness** is any nonnegative integer and will result in a thicker line.

- **rline x1;y1;color;thickness** is similar to the **line** command, but a line is drawn from the endpoints of the last line drawn to **(xold+x1,yold+y1)** which becomes then new last point. All other options are the same. This is thus a "relative" line.

- **rect x1;y1;x2;y2;color;thickness** draws a rectangle with lower corner **(x1,y1)** to upper corner **(x2,y2)** with optional color and thickness.

- **frect x1;y1;x2;y2;color** draws a filled rectangle with lower corner **(x1,y1)** to upper corner **(x2,y2)** with optional color.

- **circ x1;y1;rad;color;thick** draws a circle with radius **rad** centered at **(x1,y1)** with optional color and thickness.

- **fcirc x1;y1;rad;color** draws a filled circle with radius **rad** centered at **(x1,y1)** with optional color.

- **ellip x1;y1;rx;ry;color** draws an ellipse with radii **rx,ry** centered at **(x1,y1)** with optional color and thickness.

- **fellip x1;y1;rx;ry;color** draws a filled ellipse with radii **rx,ry** centered at **(x1,y1)** with optional color.

- **comet x1;y1;type;n;color** keeps a history of the last `n` points drawn and renders them in the optional **color**. If `type` is non-negative, then the last n points are drawn as a line with thickness in pixels of the magnitude of **type**. If **type** is negative, filled circles are drawn with a radius of **-thick** in pixels.

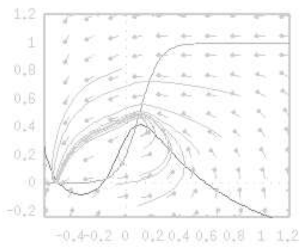
- **text x1;y1;s** draws a string **s** at position **(x1,y1)** with the current color and text properties. Only the coordinates can depend on the current values.

- **vtext x1;y1;s;z** draws a string **s** followed by the floating point value **z** at position **(x1,y1)** with the current color and text properties. Thus, you can print out the current time or value of a variable at any given time.

As with ODE files, it is possible to create arrays of commands using the combination of the [**i1..i2**] construction. For example

```
fcircle [1..9]/10;.5;.02;[j]/10
```

will create 9 circles centered at 0.1, 0.2, and so on, with radius 0.02, and color 0.1, 0.2, respectively.



Chapter 9

Tricks and advanced methods.

9.1 Introduction

In this chapter, I will describe a bunch of tricks and other advanced features that are not used that often but can be a great deal of help on those occasions when they are needed.

9.2 Graphics tricks

9.2.1 Better plots

Exporting data

If you want to get real high quality graphics, say, for publication, with multiple graphs and figures within figures, then you should probably export the data to an ASCII file and use a plotting program such as `xmgr` or `excel`. Most of these programs will read a series of (x,y) values that are in space delimited format in a file. For example, if you have an ODE file with two curves plotted on it, say $V(t), w(t)$ as functions of t, then click on `Graphic stuff Export` and give a file name. The resulting file will have three columns t, V, w separated by spaces. You can read these in and plot them in your favorite graphing program.

You can export the entire simulation data set by clicking the `Write` button in the **Data Viewer** .

In the **AUTO window** you can write data from the current view by clicking on `File Write pts` . *XPPAUT* can read this data and interpret it so that you can place a color bifurcation diagram into the *XPPAUT* main window for labeling, etc. The file consists of five columns of data `x,y1,y2,n,b`. The first three coordinates are the x coordinate of the current view, and the maximum and minimum y values respectively. $n = 1, 2, 3, 4$ for stable fixed points, unstable fixed points, stable periodic orbits, or unstable periodic orbits. b is the branch number. More complete information about a bifurcation diagram is obtained by using the `File All info` command. The result is a series of rows of numbers with the following information: $(n, b, p_1, p_2, T, y_1^{hi}, \dots, y_N^{hi}, y_1^{lo}, \dots, y_N^{lo}, r_1, m_1, \dots, r_N, m_N)$ where n is the type of point, b is the branch number, p_1, p_2 are the active parameters, T is the period, y^{hi}, y^{lo} are all the max and min values of the coordinates of the system, and (r_j, m_j) are the real and imaginary parts of the eigenvalues at that point. So, for example, if you had a 3 dimensional system and you wanted to plot the real part of the eigenvalues as a function of the main parameter, you would want to plot the third column p_1 and the 12th, 14th, and 16th columns, r_1, r_2, r_3.

You can also import data to *XPPAUT* using the `Read` button in the **Data Viewer** . Each column goes into the associated column in the **Data Viewer** . If there are more columns in the data file than in the **Data Viewer** , the first k columns will be read in, where k is the number of columns in the **Data Viewer** . I often read in data from PDE simulations so that I can use the animation features of *XPPAUT* .

Text etc.

XPPAUT has some limited capabilities for putting text, lines, and symbols in your graph. These are found under the menu item `Text etc` . You can put text of five different sizes and two different fonts in the figure. The fonts are Times-Roman and Symbol. The latter allows you to put Greek letter, but you have to know the corresponding Latin letter. E.g. $a = \alpha$ is obvious, but $q = \theta$ is not so obvious! You can mix fonts and have sub- and superscripts in the text that you write. To do this, use the five escape sequences:

\0 Use the normal Times-Roman font

\1 Use the Symbol font

\s Subscript

\S Superscript

\n Normal - no sub- or superscripts

\**{** `expr` **}** evaluate the expression `expr` before rendering the text.

Thus if you type in `\1q\0\sj\n = a+b\S2\n+\1b` then the output will be:

$$\theta_j = a + b^2 + \beta$$

since “q” becomes θ in the Symbol font and “b” becomes β.

If you want a text expression to be evaluated and permanently set so that when the quantity `expr` changes, the text will not, then preface the string with the % symbol. For example, consider the two text strings, `%I=\{iapp}`, and `I=\{iapp}`. Both will appear the same. However, if you change the parameter `iapp`, then the second one will reflect this change while the first remains frozen with the original value.

The `Arrow` option lets you draw little arrowheads in the direction that you define with the mouse. The tip of the arrow head is the first point and the direction is defined by the release point of the mouse. You will have to play around with this until you are happy with it as the relationship between the `size` and what is drawn is obscure. The `Pointer` option lets you draw straight lines with arrow heads. Choose size 0 to put on no arrow heads. The `Marker` option lets you draw a variety of symbols all centered at the position of the mouse when you click it. The `markerS` option lets you mark points along the trajectory you have drawn. In addition to size, shape, and color, you should choose the starting row (row 0 starts at the beginning of the trajectory, row 1 is at the next output point, end so on), the number of markers, and the number of rows to skip between each. You can alter the little graphics objects or delete them all from this menu item.

Line type

In the `Graphic stuff Add curve` or `Edit curve` dialog box, you can choose both a color and a linetype. Linetype 1, the default, is a solid line. Linetype 0 draws a single point instead of a line. However, a negative line type, say, -3, will draw a small circle of radius 3 points at each point in the trajectory. Negative linetypes are useful for displaying periodic orbits of maps since they are easier to see than single points on the screen.

Problems with three-d plots

Sometimes, three-d plots seem to cut off the axes. You can rectify this by clicking on the `Window Window` command and making the window for `xlo,xhi,ylo,yhi` a bit bigger.

9.2.2 Plotting results from Range Integration

XPPAUT is a computer program. It is thus, pretty dumb and can be tricked into doing what you want it to even though it was not meant to. Suppose you have a system of equations and you want to keep the results from a range integration (`Initialconds Range`). If you have fewer than 26 different curves, then you can click on `Graphic stuff Freeze On freeze` and then do the range integration. Each curve will be kept and can be plotted or output to Postscript. This is probably the best way to keep the results.

There is another way which doesn't limit the number of curves and does not use up the frozen curves. Here is the trick. In the ODE file, make the maximum amount of storage large, e.g., in the ODE file, add the line `@ maxstor=20000` so that 20000 points will be stored rather than the default of 4000. Run the file. In the `Initialconds Range` dialog box, type in `No` in the `Reset storage` box and run the range integration. All the data from each of the runs is concatenated into one long stream. The problem is that when you plot the results, a line from the end of one run is drawn to the beginning of the next. This can look quite ugly. There are two ways to avoid this. The first is to make the linetype 0 or negative.

But this is not completely satisfactory since you just get points instead of nice solid lines. The second is to take advantage of a plotting trick *XPPAUT* uses. When *XPPAUT* plots solutions to equations defined on a torus, it "knows" that there is a jump from T to 0 where T is the period of the torus. (See section 9.5.4.) As a consequence, it looks for jumps and will not plot lines joining points that are more than some fraction of this apart in distance. Thus, look at your graph and estimate how big the jump from the end of one run to the beginning of the next is. Say, for example, you are plotting many runs against time and the time interval is 20. **AFTER** you have done your integration click on the `phAsespace` and choose `All` . For the `Period` choose a number slightly less than this jump. *XPPAUT* will see that the distance between these two points is larger than the "period" and will not plot line between them. Furthermore, the PostScript plots will also not have this jump.

9.3 Fitting a simulation to external data.

XPPAUT is able to import external data, solve an initial value problem and used a least-squares approach to find the best values of initial data and parameters to match the data. The method used is the Marquardt-Levenberg algorithm (see Numerical Recipes in C). The implementation I have used here is somewhat restrictive and every parameter and initial condition is given equal weight. Furthermore, there are no constraints on the parameters or initial data. I will give an example of how to use this on some model data from an enzymatic reaction. The differential equations are

$$\begin{aligned} c' &= k_1(a_0 - c - 2d)(b_0 - c - 2d) - k_2 c - 2(k_3 c^2 - k_4 d) \\ d' &= k_3 c^2 - k_4 d \end{aligned}$$

and the data present 11 equally spaced values of (c, d) at $t = 0, \ldots, 70$. The data file is called `mod.dat` and here it is

```
0    0        0
7    1.065    0.0058
14   1.383    0.2203
21   0.9793   0.4019
28   1.107    0.3638
35   0.7289   0.456
42   0.7236   0.5014
49   0.4674   0.715
56   0.6031   0.4723
63   0.6149   0.7219
70   0.3369   0.7294
```

Suppose that you have a vague idea of what the parameters are, but want to do better. Here is the ODE file for this problem, `kinetics.ode`

```
# kinetics.ode
# simple enzyme model to fit some data
c'=k1*(a0-c-2*d)*(b0-c-2*d)-k2*c-2*(k3*c*c-k4*d)
d'=k3*c*c-k4*d
init c=0,d=0
par a0=2,b0=3,k1=.02,k2=.002,k3=.02,k4=.004
@ total=70
done
```

Let's first see how the data and the model compare before running the least squares. Run *XPPAUT* on this file. Don't integrate the equations yet. Click on `Load` in the **Data Viewer** and load the data `mod.dat` into *XPPAUT* . Plot d vs time and Freeze this in color 1. Plot c vs time and Freeze this in color 0. Now integrate the equations. `Graphic stuff` `Edit crv` and change the color to color 8, and click on `Ok` . `Graphic stuff` `Add Crv` and make `Y-axis:d` and `Color:9` in the dialog box. Window the figure so that all curves are visible. You will see 4 curves; two are the data curves and two are the components of the simulation. This is the **before** picture. Now, we will run the curve fit to see if we can do better. Click on `nUmerics` `stochastic` `fIt data` and you will get a large dialog box. Fill it in as follows:

`File:` mod.dat	`Ncols:` 3
`Fitvar:` c,d	`To Col:` 2,3
`Params:` a0,b0,k1,k2,k3,k4	`Params:`
`Tolerance:` 0.001	`Epsilon:` 1e-5
`Npts:` 11	`Max iter:` 20

This tells *XPPAUT* the name of the file, how many columns the file has, the names of the variables you want to fit to their respective columns. (Thus `c` is fit to column 2 and `d` to column 3.) You also tell *XPPAUT* the names of the parameters that will be varied. The `tolerance` is used to determine when further changes are no longer necessary. Let χ_1, χ_2 be two successive least square values. If either $|\chi_1 - \chi_2|$ or $|\chi_1 - \chi_2|/\chi_2$ are less than the tolerance, then the program assumes success. Don't make this too small as otherwise, you are wasting time. The parameter `Epsilon` is used in numerical derivatives. The parameter `Max iter` sets the maximum number of iterates to try to achieve the desired tolerance. Finally `Npts` is the number of data points (rows) that you want to use. *XPPAUT* uses the first column of data in the data file to determine the time points of the simulation. These should be in increasing order but are not required to be equally spaced.

Once you have put the values into the dialog box, click on `Ok` and a bunch of stuff will flash by in the terminal window. After a few seconds, a message box should pop up that says `Success`. This means convergence was achieved and a minimum was found. However, notice that in the **Parameter Window** , two of the kinetic constants, k_2, k_4 are negative. This is non-physical. So, what can one do about this? The easiest strategy is to set them both to zero or some very small positive number. Then redo the fit only this time do not include them as parameters in the fitting algorithm – they will be left alone. Click on `stochastic` `fIt data`

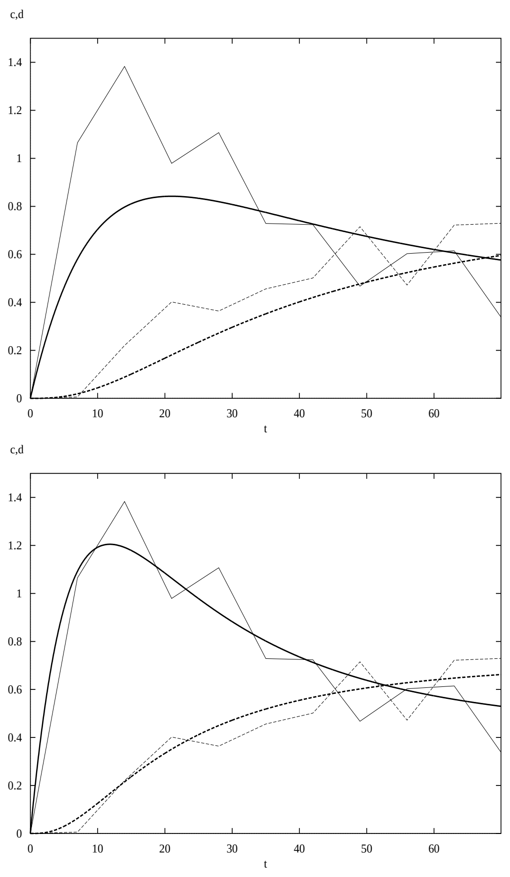


Figure 9.1. *Curve fit to experimental kinetics model. Top show the initial guess and bottom shows the parameters found by curve fitting.*

again and edit the `Params` entry by deleting `k2,k4` from the list and clicking `Ok` . Again, the fit will run successfully. This time no physical constraints are violated. `Escape` from the numerics menu and click on `Initialconds Go` and look at the new solutions you have computed. They are much better. Figure 9.1 shows the results of the initial and final choices of parameters.

9.4 The data browser as a spreadsheet.

The **Data Viewer** has a number of interesting features that allow you to manipulate data in the columns much like you would in a spread sheet. For example, sometimes you might want to numerically differentiate or integrate data in a column or look at a logarithmic plot. This is all possible by manipulating columns in the **Data Viewer** . So, let's look at some of the functionality of the data browser. Run the Morris-Lecar equation model, `ml.ode`. In the **Data Viewer** , click on `Add` `col` . For the name, type in `il`, an abbreviation for the leak current I_L. For the formula, type in `gl*(v-el)` and then click on `Add it` . A new column will appear in the data browser. You can plot this new quantity and when you integrate the equations, this new quantity changes appropriately. You can add more quantities as well. The `Del Col` does not work properly so that it has been inactivated. If you make a mistake in the definition of the formula for the new quantity, you can always edit it using `File Edit RHS's` in the **Main Window** .

If you don't want to add new column but instead just want to look at, say the log of a quantity for that particular run, you can always click the `Replace` button in the **Data Viewer** . This prompts you for the name of the column and then the formula. The `Unreplace` button undoes the most recent replacement.

You can numerically integrate or differentiate a column of data using the symbols `&` and `@` respectively. Thus, in the Morris-Lecar file, suppose, you wanted the derivative of the voltage. Then replace the voltage by first choosing `v` as the column to replace and then typing in `@v` for the formula. Then you can plot the derivative of the voltage. Note that differentiation and integration are **post integration** commands only. That is, you cannot use these in a new column formula or right-hand side.

Occasionally, you need a sequential list of numbers. You can replace a column with such a list. The formula `a:b` creates a list of numbers the same length as the number of rows in the **Data Viewer** , starting with `a` and ending with `b`. The formula, `a;b` creates a list starting with `a` and incrementing by `b` for each row. For example, replace `t` with `0:6.283` and then replace `V` with `sin(t)`. Plot `V` versus t and you will see a nice sine wave! Pretty lame, huh.

One trick is to read in data from a file and manipulate it with the data browser and use *XPPAUT* as a fancy graphing program.

9.5 Oscillators, phase models, and averaging.

One of the main areas of my own research concerns the behavior of coupled nonlinear oscillators. In particular, I have been interested in the behavior of weakly coupled oscillators. Before describing the features of *XPPAUT* that make such analyses easy, I will discuss a bit of the theory. Consider an autonomous differential equation:

$$X' = F(X)$$

which admits as a solution an asymptotically stable periodic solution, $X_0(t) = X_0(t+P)$ where P is the period. Now suppose that we couple two such oscillators, say, X_1 and X_2:

$$X_1' = F(X_1) + \epsilon G_1(X_2, X_1)$$
$$X_2' = F(X_2) + \epsilon G_2(X_1, X_2)$$

where G_1, G_2 are possibly different coupling functions and ϵ is a small positive number. One can apply successive changes of variables and use the method of averaging to show that for ϵ sufficiently small,

$$X_j(t) = X_0(\theta_j) + O(\epsilon),$$

with

$$\theta_1' = 1 + \epsilon H_1(\theta_2 - \theta_1) \quad \theta_2' = 1 + \epsilon H_2(\theta_1 - \theta_2),$$

and H_j are $P-$periodic functions of their arguments. This type of model is called a **phase model** and has been the subject of a great deal of research. One of the key questions is what is the form of the functions H_j and how can one compute them? This is where *XPPAUT* can be used. The formula for H_j is

$$H_j(\phi) \equiv \frac{1}{P}\int_0^P X^*(t)\cdot G_j[X_0(t+\phi), X_0(t)]\,dt \tag{9.1}$$

which is the average of the coupling with a certain $P-$periodic function, X^* called the **adjoint** solution. This equation satisfies the linear differential equation and normalization:

$$\frac{dX^*(t)}{dt} = -[D_X F(X_0(t))]^T X^*(t), \quad X^*(t)\cdot X_0'(t) = 1.$$

Here, $D_X F$ is the derivative matrix of F with respect to X and A^T is the transpose of the matrix A.

Thus, to compute the functions H_j, the adjoint must be computed along with the integral. *XPPAUT* implements a numerical method for computing the adjoint, $X^*(t)$ invented by Graham Bowtell. The method only works for oscillations which are asymptotically stable. Here is the idea. Let $B(t) = (D_X F(X_0(t)))$ Since the limit cycle is asymptotically stable, if we integrate the equation

$$Y' = B(t)Y$$

forward in time, it will converge to a periodic orbit proportional to $X_0'(t)$ as a consequence of the stability of the limit cycle and the translation invariance. Similarly, the solution to

$$Z' = -B(t)^T Z$$

will converge to a periodic orbit if we integrate it *backward* in time. As you can see, this is the desired adjoint solution and is how *XPPAUT* computes it. Once $X^*(t)$ is computed, it is trivial to compute the integral and thus compute the interaction function.

9.5.1 Computing a limit cycle and the adjoint

To use *XPPAUT* to compute the adjoint, a necessary first step to computing the interaction function H, we first have to compute exactly one full cycle of the oscillation. Let's use as our example the Morris-Lecar equation:

$$\begin{aligned} v' &= I + g_l(e_l - v) + g_k w(e_k - w) + g_{ca} m_\infty(v)(e_{ca} - v) \\ w' &= (w_\infty(v) - w)\lambda_w(v) \end{aligned}$$

We will look at:

$$\begin{aligned} v_1' &= f(v_1, w_1) + \epsilon(v_2 - v_1) \\ w_1' &= g(v_1, w_1) \\ v_2' &= f(v_2, w_2) + \epsilon(v_1 - v_2) \\ w_2' &= g(v_2, w_2) \end{aligned}$$

Since the method of averaging depends on the two oscillators being identical except possibly for the coupling, all you need to do is integrate the isolated oscillator. Use the file `ml.ode`. Change the parameters, `I=.09, phi=0.5`. Integrate the equations and click on `Initialconds Last` a few times to make sure you have gotten rid of all transients. Now you are pretty much on the limit cycle. To compute the adjoint, you need one full period. In many neural systems, coupling between oscillators occurs only through the voltage and thus, the integral (9.1) involves only one component of the adjoint, the voltage component. For whatever reason, the numerical algorithm for computing the adjoint for a given component converges best if you start the oscillation at the *peak* of that component. Thus, since we will only couple through the potential in this example, we should start the oscillation at the peak of the voltage. Here is a good **trick** for finding that maximum. In the **Data Viewer** click on `Home` which makes sure that the first entry of the data is at the top of the **Data Viewer** . Click on `Find` and in the dialog box, choose the variable `v` and for a value, choose `1000` and then `Ok` . *XPPAUT* will find the closest value of `v` to `1000` which is obviously the maximum value of `v`. Now click on `Get` in the **Data Viewer** which grabs this as initial conditions. In the **Main Window** , click on `Initialconds Go` to get a new solution. Plot the voltage versus time. Use the mouse to find the time of the next peak (With the mouse in the graphics window, click the button and move the mouse around. At the bottom of the windows, read off the values). The next peak is at about 22.2. In the **Data Viewer** scroll down to this time and find exactly where `v` reaches its next maximum. Note that it is, as we suspected, at $t = 22.2$. In the **Main Window** , click on the `nUmerics` menu and then `Total` to set the total integration time. Choose `22.21` (it is always best to go a little bit over - but just a little.) Escape to the main menu and click on `Initialconds Go` . Now we have one full cycle of the oscillation! The rest is easy.

Computing the adjoint

Click on `nUmerics Averaging New adjoint` and after a brief moment, *XPPAUT* will beep. (Sometimes, when computing the adjoint, you will encounter the `Out of Bounds` message. In this case, just increase the bounds and it recompute the adjoint.) Click on `Escape` and plot v versus time. This time, the adjoint of the voltage is in the **Data Viewer** under the voltage component. (Note that the adjoint is almost strictly positive and looks nearly like $1 + \cos t$. This is no accident and has been explained theoretically.)

9.5.2 Averaging

Now we can compute the average. Recall that the integral depends on the adjoint, the original limit cycles and a phase-shifted version of the limit cycle:

$$X^*(t) \cdot G(X_0(t+\phi), X_0(t)).$$

In *XPPAUT* , you will be asked for each component of the function G. For unshifted parts, use the original variable names, e.g., `x,y,z` and for the shifted parts, use primed versions, `x',y',z'`. The coupling vector in our example is

$$(V(t+\phi) - V(t), 0).$$

Thus, for our model, the two components for the coupling are `(v'-v, 0)`. This says that we take the phase-shifted version of the variable `v`, called `v'` by *XPPAUT* and subtract from it the unshifted version of `v`. With these preliminaries, it is a snap to compute the average. Click on `nUmerics Averaging Make H` . Then type in the first component of the coupling, `v'-v` and the second `0`. Then let it rip. In a few moments, the calculation will be done. If your system has more than two columns in addition to time, `t` (as this example does - `v,w, ica, ik`) then the first column contains the average function $H(\phi)$, the second column contains the odd part of the interaction function, and the third column contains the even part. Plot the function H by escaping back to the main menu and plotting `v` versus time – remember `v` is the first column in the browser after the `t` column. This is a periodic function. For later purposes, we want to approximate this periodic function. Click on `nUmerics stocHastic Fourier` and choose `v` as the column to transform. Look at the data browser and observe the first three cosine terms (column 1 after the `t` column) and the first three sine terms (column 2). They are (3.34, -3.05, -.29) and (0,3.61,-0.33) respectively. Thus to this approximation

$$H(\phi) = 3.34 - 3.05\cos\phi - 0.29\cos 2\phi + 3.61\sin\phi - 0.33\sin 2\phi. \tag{9.2}$$

To get the interaction function, adjoint, or original orbit back into the data browser, click on `Averaging Hfun` etc from the `nUmerics` menu.

Summarizing,

1. Compute exactly one period of the oscillation – you may want to phase-shift it to the peak of the dominant coupling component.
2. Compute the adjoint from the `nUmeric Averaging` menu.
3. Compute the interaction function using the original variable names for the unshifted terms and primed names for the shifted terms.

9.5.3 Phase response curves

One of the most useful techniques available for the study of nonlinear oscillators is the **phase response curve** or PRC for short. The PRC is readily computed experimentally in real biological and physical systems. The PRC is defined as follows.

Suppose that there is a stable oscillation with period T. Suppose that at $t = 0$ one of the variables reaches its peak. (This can always be done by translating time.) At a time τ after the peak, one of the state variables is given a brief perturbation that takes it off the limit cycle. This will generally cause the next peak to occur at a time $T'(\tau)$ that is different from the time it would have peaked in absence of the perturbation. The PRC, $\Delta(\tau)$ is defined as

$$\Delta(\tau) \equiv 1 - T'(\tau)/T.$$

If $\Delta(\tau) > 0$ for some τ we say that the perturbation advances the phase since the time of the next maximum is shortened by the stimulus. Delaying the phase occurs when $\Delta(\tau) < 0$. Experimental biologists have computed the PRCs for a variety of systems such as cardiac cells, neurons, and even the flashing of fireflies. We will use *XPPAUT* to compute the PRC for the van der Pol oscillator:

$$\ddot{x} = -x + \dot{x}(1 - x^2)$$

subjected to a square wave pulse of width σ and amplitude a.

Here is how we will set up the problem. We will let τ be a variable rather than a parameter so that we can range through it and keep a record – it is like a bifurcation parameter. We will define a parameter T_0 which is the unperturbed period. We will use the ability of *XPPAUT* to find maxima of solutions to ODEs and have the computation stop when the maximum of x is reached. The time at which this occurs is the time $T'(\tau)$ and so we will also track an auxiliary variable $1 - t/T_0$ which is the PRC. Here is the ODE file:

```
# vdpprc.ode
# PRC of the van der Pol oscillator
init x=2
x'=y
y'=-x+y*(1-x^2)+a*pulse(t-tau)
tau'=0
pulse(t)=heav(t)*heav(sigma-t)
par sigma=.2,a=0
par t0=6.65
aux prc=1-t/t0
@ dt=.01
done
```

Note that the function `pulse(t)` produces a small square-wave pulse. To compute the PRC, you want to integrate the equations for a bit to get a good limit cycle with no perturbation ($a = 0$). Then start at the maximum value of x and integrate until the next maximum. This will tell you the base period. Then you want to apply the perturbation at different times and compute the time of the next maximum and then from this get the PRC. Fire up *XPPAUT* with this file. Follow these simple steps:

1. Integrate it and then integrate again using the `Initialconds Last` (**I L**) command to be sure you have integrated out transients.

2. Now find the the variable of interest, in this case, x. To do this, in the **Data Viewer** click on `Home` to go to the top of the data window. Click on `Find` and type in `x` for the variable and `100` for the value. *XPPAUT* will try to find the value of `x` closest to 100. This will be the maximum. Click on `Get` to load this as an initial condition.

3. Figure out the unperturbed period. One way is to integrate the equations and estimate the period. A better way is to let *XPPAUT* do it for you. In the **Main Window** , click on `nUmerics Poincare map Max/Min` and fill in the dialog box as follows:

`Variable:` x
`Section:` 0
`Direction:` 1
`Stop on sect:` Y

   and then click on `Ok` . You have told *XPPAUT* to integrate, plotting out only the maxima (Direction=1) of x. The `Stop on section` ends the calculation when the section is crossed. Click on `Transient` and choose 4 for the value. This is so we don't stop at the initial maximum. `Transient` allows the integrator to proceed for a while before looking at and storing values. Now exit the numerics menu by clicking `Esc` . Click on `Initialconds Go` and the program will integrate until x reaches a maximum. In the **Data Viewer** , click on `Home` and you should see that `Time` has a value of around 6.6647. This is the unperturbed period. Set the parameter `t0` to the value in the **Data Viewer** .

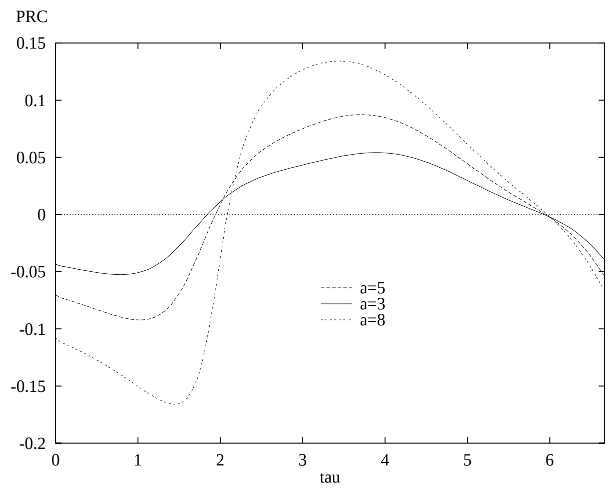
4. Compute the PRC. Now we are all set to compute the PRC. Change the perturbation amplitude from 0 to 3. Click on `Initialconds Range` and fill in the dialog box as follows:

`Range over:` tau
`Steps:` 100
`Start:` 0
`End:` 6.6647
`Reset storage:` N

   and click `Ok` . It should take a second or two.

5. Plot the auxiliary quantity `PRC` against the variable `tau`. (Click on `Viewaxes 2D` and put `tau` on the `X-axis` and `PRC` on the `Y-axis`. Click `OK` and then `Window Fit` to let *XPPAUT* figure out the window.) You should see something the the first curve in Figure 9.2

Freeze this curve and try using a different value of the amplitude `a`.



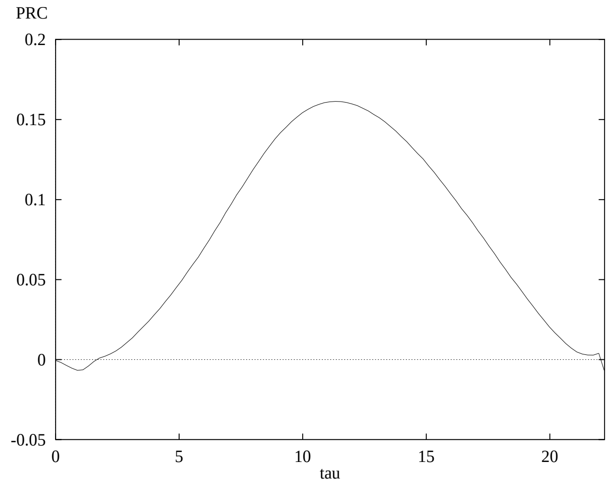


Figure 9.2. *Top: The PRC for the van der Pol oscillator with different amplitudes. Bottom: The PRC for the Morris-Lecar model.*

Exercise. Compute the PRC for the Morris-Lecar oscillator by adding the required parts to the file `ml.ode`. Define a square-wave pulse just like above with a width of 0.25 and a magnitude of 0.1. If you are stuck, download the file `mlprc.ode` which sets up the ODE for you. You should get something like the bottom plot of figure 9.2.

9.5.4 Phase models

In the previous section we saw how to reduce a pair of coupled oscillators to a pair of scalar models in which each variable lies on a circle. *XPPAUT* has a way of dealing with flows on systems of scalar variables each of which lie on a circle. The phase-space of a system of, say, two such variables is the two-torus. Let's start with the simplest phase model:

$$\theta_1' = \omega_1 + a\sin(\theta_2 - \theta_1)$$
$$\theta_2' = \omega_2 + a\sin(\theta_1 - \theta_2)$$

representing a pair of sinusoidally coupled oscillators with different uncoupled frequencies,ω_1, ω_2 and coupling strength, a. The ODE file for this is straightforward:

```
# phase2.ode
# phase model for two coupled oscillators
th1'=w1+a*sin(th2-th1)
th2'=w2+a*sin(th1-th2)
par w1=1,w2=1.2,a=.15
@ total=100
done
```

Fire this up and integrate the equations. You will get the `Out of bounds` message at $t = 90$ or so. That is because the variables θ_j are actually defined only modulo 2π and are not really going out of bounds but instead just wrapping around the circle. Thus, it is more proper to look at θ_j modulo 2π. If define an auxiliary variable which is θmod 2π and then plot it, you will get a series of ugly lines that cross from one part of the screen to the other. This is because *XPPAUT* does not know that this particular variable is defined on the circle. There is a simple way to tell *XPPAUT* which of the state variables lie on the circle. Click on `phAsespace Choose` (**A C**) and when prompted for the period, choose the default which is approximately 2π. A little window will appear with all of your variables included. Move the cursor to the left of each of them and click the mouse to see a little `X` next to the variable. Put this mark next to both variables and click on done. This tells *XPPAUT* that these are both considered "folded" variables and will be folded mod 2π. Now reintegrate the equations. This time you get no such out of bounds message – instead you will see that the plots are all modulo 2π. Switch the view to the phaseplane for the two variables, θ_1, θ_2. (In the **Initial Data Window** click on the box to the left of each variable and then on the `xvsy` button.) You will see that the trajectories seem to converge on a diagonal line which is slightly shifted upward. Try integrating using the `Initialconds mIce` (**I I**) command to choose a variety of initial conditions. They should converge to the same line. This is an example of a phase-locked solution – an invariant circle on the torus. Change a from 0.15 to 0.08 and integrate the equations again. Notice how the trajectories do not converge to an attractor. Instead, the whole torus will gradually fill up. Phase-locking no longer occurs.

A derived example

Now let's turn to the example that we derived in the previous section and see if a pair of oscillators will phaselock when we use the interaction function defined in (9.2), `phase_app.ode`:

```
# phase_app.ode
# phase model for two coupled oscillators
# using numerically computed H
h(x)=3.34-3.05*cos(x)-.29*cos(2*x)+3.61*sin(x)-.33*sin(2*x)
th1'=w1+a*h(th2-th1)
th2'=w2+a*h(th1-th2)
par w1=1,w2=1,a=.1
@ total=100
@ fold=th1,fold=th2
@ xlo=0,ylo=0,xhi=6.3,yhi=6.3,xp=th1,yp=th2
done
```

I have added a few new `@` commands. The `fold=th1` directive tells *XPPAUT* to make a circle out of the variable `th1` that is to mod it out. All variables that you want to mod out can be set by the `fold=name` directive. This automatically tells *XPPAUT* to look for modded variables. The default period is 2π so we don't have to change it. If you want to change the period to say, 3, set it with the command `@ tor_period=3`. Run *XPPAUT* on this using the mouse to set some initial conditions. Note how all initial data synchronizes along the diagonal implying $\theta_1 = \theta_2$. Change the intrinsic frequency, `w2` and see how big you can make it before phase-locking is lost.

Pulsatile coupling

Winfree (1967) introduced a version for coupling oscillators using the phase-response curve of the oscillator assuming that the interaction between oscillators was only through the phase and took the form of a product. Ermentrout and Kopell (1991) proved that this was a reasonable model when certain assumptions were made concerning the attractivity of the limit cycle. Here is a pair of pulse-coupled phase models:

$$\frac{d\theta_1}{dt} = \omega_1 + P(\theta_2)R(\theta_1)$$
$$\frac{d\theta_2}{dt} = \omega_2 + P(\theta_1)R(\theta_2).$$

Think of $R(\theta)$ as the phase-response curve of the oscillator and $P(\theta)$ as the pulse-coupling. For example if $P(\theta)$ is a Dirac delta-function, then this model reduces to a one-dimensional map. This instance of coupling is simulated in 9.6.2. The following ODE is a case in which the coupling is through a smooth function.

```
# phasepul.ode
# pulsatile phase model
r(x)=-a*sin(x)
p(x)=exp(-beta*(1-cos(x)))
par a=3,beta=5
x1'=w1+p(x2)*r(x1)
x2'=w2+p(x1)*r(x2)
```

```
par w1=1,w2=3
@ total=100
@ fold=x1,fold=x2
@ xp=x1,yp=x2,xlo=0,ylo=0,xhi=6.3,yhi=6.3
done
```

As in the previous example, I have set this up so that the projection is onto the torus phase-plane. Here are some things to do:

- Integrate the equations and note how solutions tend to the diagonal. There is a stable phase-locked synchronous solution.
- Change the integration time step to `-.05` and integrate the equations again. Note that there is another phase-locked solution that is out of phase. Change the integration time step back to 0.05.
- Change the coupling strength, a from 3 to -3 and repeat the first two exercises. Note how synchrony is unstable when $a < 0$.
- Change a to 3 again. Draw the nullclines. They don't intersect. Change `w1`, the frequency of the first oscillator to 0.1. Redraw the nullclines. Determine the stability of the fixed points and draw all the invariant manifolds for the saddle-point. Where do the stable manifolds go? They tend to "vertical" lines in the middle of the phase-plane. That is, there is a repelling invariant circle in which oscillator two fires repetitively and oscillator 1 just sort of wobbles around near π. All stable solutions end up at a fixed point. This is called **phase-death**.
- Now change `w1` back to 1 and change `w2` to 3. Integrate the equations. Note that for every three cycles of oscillator two, there is one cycle of oscillator one. This is an example of 3:1 phaselocking. Open another graphics window by clicking on `Makewindow New` and in this window graph both `x1` and `x2` against `t`. Window this for $80 < t < 100$ so you can see it more clearly.
- Gradually decrease `w2` toward 1 and see what other kinds of mode locking are possible. (`w1=1.9` is interesting.)

9.6 Arcana.

Here I describe a number of useful tricks that belong to no special category.

9.6.1 Iterating with fixed variables

Because "fixed" variables in *XPPAUT* are evaluated in succession, one can actually compute iterative procedures with them which will be used in the right-hand sides of the equations. In chapter 4, we saw an example of this in the Taylor series plots. Here, I present a more sophisticated example of this trick. I want to create and analyze the Morse-Thule sequence, M_n. This is a sequence of 0's and 1's. The n^{th} number in the sequence is the number of 1's in the binary expression of n modulo 2. For example, the 23^{rd} element is found by writing $23 = 10111$. This has four 1's and 4=0 modulo 2, so $M_{23} = 0$. Suppose that $n < 2^p$. Then the number of 1's in the binary expansion of n is found from the iteration:

```
s=0;
x=n;
for(i=0;i<p;i++){
s=s+mod(x,2);
x=flr(x/2);
}
```

Here `flr(x)` is the integer part of `x`. Thus, I just keep dividing x by 2 and checking if the result is odd. (This can be done much more quickly in C than I have indicated here, but this is less obscure.) After the iteration, s contains the number of 1's. We then just get M_n by modding by 2. Here is a fragment of *XPPAUT* code that implements this iteration:

```
s0=0
x0=n
%[1..16]
s[j]=s[j-1]+mod(x[j],2)
x[j]=flr(x[j-1]/2)
%
```

These are all "fixed" variables so they are evaluated at every time step. The block delimited by the `%` does the iteration 16 times so that this will work as long as $n < 2^{16} = 65536$. The quantity, `s16` has the desired sum. We have used the fact that the fixed variables are evaluated in the order in which they are defined. Here we have defined 34 fixed variables. With this bit of code, the rest of the ODE file is easy. Here is `mt.ode`:

```
# mt.ode
# the morse-thule sequence of 0's and 1's
# the sum mod 2 of the 1's in the binary expansion
# of the integers
# look at the power spectrum for z
#
init n=0
s0=0
x0=n
%[1..16]
s[j]=s[j-1]+mod(x[j],2)
x[j]=flr(x[j-1]/2)
%
n'=n+1
z'=mod(s16,2)
aux ss=s16
@ meth=discrete,total=10000,maxstor=68000
@ bound=100000
@ xlo=0,xhi=10000,ylo=0,yhi=16,yp=ss
done
```

Run this ODE file and look at the nice self-similar nature of the sums (which are plotted.) The sequence itself is just the auxiliary variable, `ss`. Now, the really cool part of this is to look at the power spectrum of the sequence, `ss`. Click on `nUmerics stocHast Power` and choose `ss` as the variable to transform. Click on `Esc` to get back to the main menu. Click on `Xi vs t` and plot `N`. You will see a very strange power spectrum which is self-similar. That is, if you blow up a region, it looks like the full spectrum. Try this with 64000 iterations and it looks essentially the same.

9.6.2 Timers

There are often situations where you may want a switch to turn on after a proscribed amount of time. For example if you have a model of a synapse that has the form:

$$ds/dt = A(t)(1-s) - s/\tau$$

where $A(t) = A_0$ is $T < t < T + h$ and zero otherwise. T is the firing time for the cell that corresponds to the synapse, s. Here is a way to do this:

```
init tf=-1000
s'=a0*heav(tf+h-t)*(1-s)-s/tau
tf'=0
global 1 v-vt {tf=t}
```

I have used the global flag to determine when the cell's voltage, `v` crosses some threshold, `vt`. At this point the variable `tf` is set to the current time `t`. The Heaviside step function switches to 1 and remains as long as `t<tf+h` and thus turns off `h` time units later.

Coupled phase response curves

Related to this issue of tracking time, is keeping a record of interspike intervals and/or relative firing times in a system of neural oscillators. For simplicity, consider two coupled neural oscillators, say, coupled by a phase-response curve (see section):

$$x_1' = \omega_1 + \delta(x_2)P_{21}(x_1) \quad x_2' = \omega_2 + \delta(x_1)P_{12}(x_2).$$

We assume that whenever x_j crosses 1, it is reset to zero and $x_k = x_k + P(x_k) \equiv F(x_k)$. We also assume that P is 1-periodic and vanishes at $x = 0$. Then, we can write the ODE file for this as

```
# map2.ode
# pair of coupled maps
x1'=w1
x2'=w2
p(x)=-a*sin(2*pi*x)
f(x)=mod(x+p(x),1)
```

```
par a=.05
par w1=1,w2=.9
global 1 x1-1 {x1=0;x2=f(x2)}
global 1 x2-1 {x2=0;x1=f(x1)}
@ dt=.0101
done
```

The strange value of `dt` is due to a problem with the `global` command which occasionally fails if the test condition is satisfied exactly, i.e., `x1=1` at a particular time step.

Now, this particular ODE file gives the values of the state variables, but not the timing difference between the two cells nor the firing time interval between them. Let ϕ be the time difference between the time that cell 1 fires and cell 2 fires. Let T_j be the interval between successive firings of the respective cells. Then we can track these keeping in mind that the `global` command only allows you to modify state variables. We introduce 5 new variables, `t1f,t2f, t1,t2, phi` all of which satisfy $u' = 0$. Here is the new ODE file

```
# map2.ode
# pair of coupled maps
# and timing info
x1'=w1
x2'=w2
t1f'=0
t2f'=0
t1'=0
t2'=0
phi'=0
p(x)=-a*sin(2*pi*x)
f(x)=mod(x+p(x),1)
par a=.05
par w1=1,w2=.9
global 1 x1-1 {x1=0;x2=f(x2);t1=t-t1f;t1f=t}
global 1 x2-1 {x2=0;x1=f(x1);phi=t-t1f;t2=t-t2f;t2f=t}
@ dt=.0101,total=50, bound=10000
done
```

Each time x_1 fires, I update the interval between the last firing, `t1=t-t1f` and then update the firing time of 1, `t1f=t`. Each time that x_2 fires, I update similar intervals but also the timing difference, `phi=t-t1f`. Try this file with the parameters as in the file. Plot `t1,t2` on the same plot. Plot `phi`. Increase `a` to 0.06. Try this again. Change `a` to -0.06 and solve it again. Finally, set `a=0.1` and `w2=0.53` and integrate the equations. Look at `t1,t2` and interpret what this means.

9.6.3 Initial data depending on parameters

One unfortunate shortcoming in *XPPAUT* is the inability of the program to set initial data that depend on parameters. Parameters can be defined in terms of other parameters with the

```
!<name>=<formula>
```

declaration. However, there is no easy way to define initial data in terms of a parameter. In spite of this shortcoming, there is a little trick that you can use. This trick exploits the fact that `global` flag conditions can set the state variables to anything. Let's consider the classic problem of firing a cannonball. The parameter that you want to vary is the angle of the cannon, θ. Thus, the differential equation is

$$m\ddot{x} = -f\dot{x}, \qquad m\ddot{y} = -mg - f\dot{y}$$

with

$$x(0) = y(0) = 0, \quad \dot{x}(0) = v_0 \cos\theta, \quad \dot{y}(0) = v_0 \sin\theta.$$

Here is the trick – we will flag $t = 0$ and immediately switch initial data. The key here is to use the global sign `0` instead of `1,-1`. The difference with using `0` is that the flag is set only when equality occurs exactly. This essentially never happens except at the start of a problem. Thus, it is a good way to set initial data that depend on a parameter. Here is the ODE file:

```
# cannon.ode
# with linear friction
x'=vx
vx'=-f*vx
y'=vy
vy'=-f*vy-g
global 0 t {vx=v0*cos(pi*theta/180);vy=v0*sin(pi*theta/180)}
par theta=30
par f=.01,g=1
par v0=2.5
init x=0,y=0
@ xlo=0,xhi=5
@ ylo=0,yhi=3
@ xp=x,yp=y,bound=100000
done
```

I have set up the problem in the $(x, y)-$plane so that you can see the trajectory of the cannon. I have also scaled the angle θ so that it is in degrees rather than radians. Try to hit the number 4 on the x-axis!

Computing basin boundaries

The ability to parametrically control initial data allows us to perform some interesting computations – we can compute the basins of attraction of systems with multiple attractors. Let's first consider a classic example, the complex cube roots of 1:

$$z^3 = 1.$$

If we rewrite this in terms of the real and imaginary parts of $z = x + iy$ then we want to solve

$$x^3 - 3xy^2 = 1 \quad 3x^2y - y^3 = 0$$

for (x, y). This represents two nonlinear equations in two unknowns. The way to solve these is to use Newton's method which we write as an iteration:

$$\begin{bmatrix} x_{n+1} \\ y_{n+1} \end{bmatrix} = \begin{bmatrix} x_n \\ y_n \end{bmatrix} - \begin{bmatrix} 3x^2 - 3y^2 & -6xy \\ 6xy & -3y^2 \end{bmatrix}^{-1} \begin{bmatrix} x^3 - 3xy^2 - 1 \\ 3x^2y - y^3 \end{bmatrix}$$

Here (x_n, y_n) is the n^{th} approximation to the root. The matrix above is the Jacobi matrix for the two functions. The point is that Newton's method produces a discrete dynamical system. There are three cube roots to 1 in the complex plane, the real root,$(x, y) = (1, 0)$, and the complex conjugate roots, $(x, y) = (1/2, \pm\sqrt{3}/2)$. Thus, we can ask the following question: Given an initial guess, (x_0, y_0) which of these roots will the guess converge to? If we color code the point (x_0, y_0) according to the root that it is attracted to, the resulting picture looks like that in figure 9.3.

This picture can be made with *XPPAUT* by taking advantage of the ability to vary initial data parametrically. Here is the ODE file used to produce the figure:

```
# cube.ode
# newtons method in the complex plane to find cube roots of 1
# z^3=1
# x^3-3*xy^2 = 1 , 3x^2y-y^3=0
# draws fractal basin boundaries
#
# tolerances
par eps=.001
# discretization of the phase-space
par dx=0,dy=0
# the three roots
par r1=1,s1=0
par r2=-.5,s2=-.8660254
par r3=-.5,s3=.8660254
# the functions
f=x^3-3*x*y^2-1
g=3*x^2*y-y^3
# the derivatives of the functions
fx=3*(x^2-y^2)
fy=-6*x*y
gx=6*x*y
gy=3*(x^2-y^2)
# the Jacobian -- used in the inverse
det=fx*gy-fy*gx
# a euclidean distance function
dd(x,y)=x*x+y*y
# if close to the root then plot otherwise out of bounds for each root
```

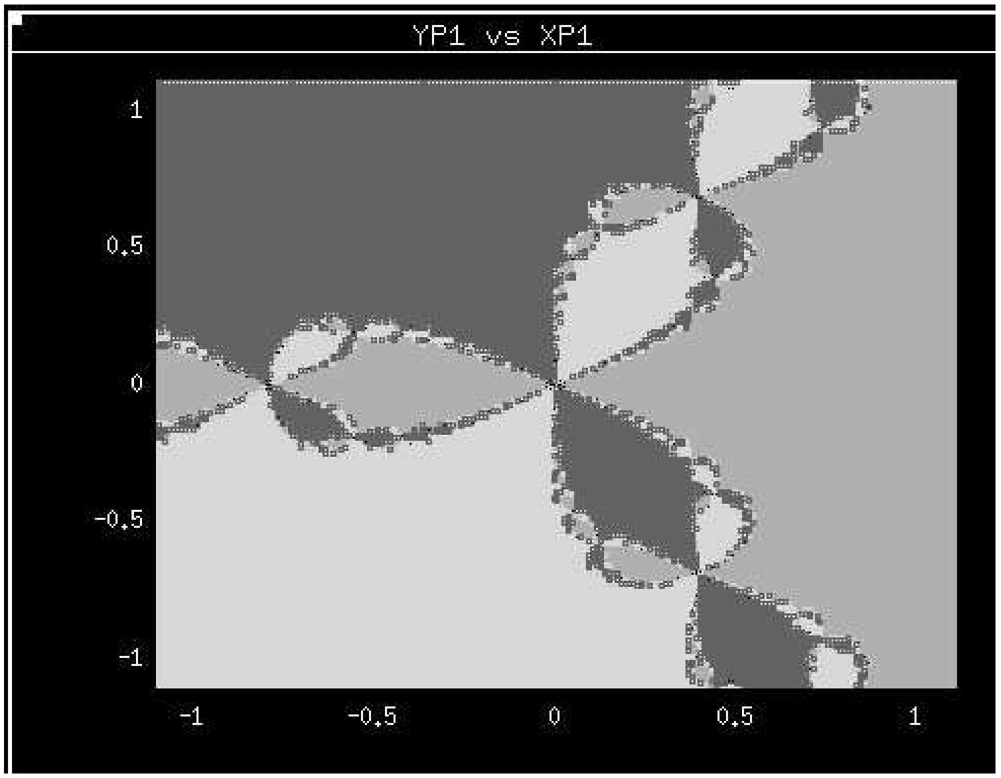


Figure 9.3. *The basin boundaries for the iteration which arises from the application of Newton's method to the equation $z^3 = 1$ in the complex plane. Each point in the plane is colored according to the root that it converges to.*

```
aux xp[1..3]=if(dd(x-r[j],y-s[j])<eps)then(x0)else(-100)
aux yp[1..3]=if(dd(x-r[j],y-s[j])<eps)then(y0)else(-100)
#
# iteration
x'=x-(f*gy-g*fy)/det
y'=y-(g*fx-f*gx)/det
# initial data
x0'=x0
y0'=y0
# set initial data as parameters
glob 0 t {x0=-1.1+dx*2.2;y0=-1.1+dy*2.2;x=-1.1+dx*2.2;y=-1.1+dy*2.2}
#
# plot all three sets of points in lousy colors
#
@ meth=discrete,total=20,bound=1000,maxstor=100000,trans=20
@ xp=xp1,yp=yp1,xp2=xp2,yp2=yp2,xp3=xp3,yp3=yp3,nplot=3,lt=-1
@ xlo=-1.1,ylo=-1.1,xhi=1.1,yhi=1.1
done
```

This rather lengthy ODE file is actually pretty simple and works as follows. We draw 3 different sets of data corresponding to the three different roots. We compute 20 iterations and only keep the last iterate by setting the total and the transient to 20. If (x_{20}, y_{20}) is close the real root, then we plot the initial conditions (x_0, y_0) other wise we plot $(-100, -100)$ which is out of the plotting range. Similarly, if the twentieth iterate is close to the first complex root, then we plot the initial condition otherwise we plot the out-of-bounds point. We treat the data for each root as three different sets of points. I hope the large number of comments make it self-explanatory. I point out the use of the `global` statement to set the initial conditions according to the values of the parameters `dx,dy`. In the options statements, I tell *XPPAUT* to plot all three pairs of points `xp1,yp1`, `xp2,yp2`, and `xp3,yp3` which are either the initial conditions of the iteration or the points (-100,-100). By choosing linetype `-1` a series of small circles will be drawn instead of dots or lines. Run this file. Click on `Initialcond 2 par range` (**I 2**) and fill in the dialog box as follows:

`Vary1:` dx	`Reset storage:` N
`Start1:` 0	`Use old ic's:` Y
`End1:` 1	`Cycle color:` N
`Vary2:` dy	`Movie:` N
`Start2:` 0	`Crv(1) Array(2):` 2
`End2:` 1	`Steps2:` 101
`Steps:` 101	`:`

and click on **Ok** . A series of colored points will be drawn. To better distinguish the points, you should click on `Graphics Edit crv` , select curve 2 and change the color from 2 which is red-orange to 7 which is green.

Exercises

- Compute the basin boundaries for the complex roots of $z^2 = 1$ and $z^4 = 1$. Note that the first is equivalent to $x^2 - y^2 = 1, 2xy = 0$ and the second is equivalent to $x^4 - 6x^2y^2 + y^4 = 1, 4(x^3y - xy^3) = 0$. For the fourth roots problem you should set the total number of iterates to 50 as well as the transient.

- Consider the discrete bistable system:

$$x_{n+1} = y_n \quad y_{n+1} = ay_b + bx_n(c^2 - x_n^2)$$

  in the same square as the above examples. There are two stable fixed points for this system $(\pm c, 0)$. Choose $a = .6, b = .3, c = .15$ and compute the basins of attraction of the two roots. You should set the total time and the transient to 100.

Invariant sets revisited

Recall that the periodically forced Duffing equation is chaotic and there can be transverse intersections of the invariant manifolds emerging from the saddle-point. The dynamics under the flow of the equation is such that points get all mixed up like cake-batter being stirred. We can see this happen as follows. Choose a region in the phase-plane. Color each point in the phase-plane black (white) if the solution with that initial data is in the $x > 0$ ($x < 0$) half-plane at some fixed time t_0. Do this for a variety of different times to see how the flow mixes everything up. This is like the basin boundaries we computed above but only for finite times. The *XPPAUT* file is like the one in the previous section but simpler.

```
# duffbas.ode
# a trick to compute the basin boundaries of the duffing equation
# better be patient it takes awhile
# here is the ODE
x'=y
y'=-.15*y+.5*x*(1-x^2)+f*cos(.8333*t)
#  dx,dy are the increment sizes, f is the forcing amplitude
par dx,dy,f=.1
# here are the initial data evaluated once per cycle
# initial data lie in the square [-2.4,2.4]x[-1.2,1.2]
!x0=-2.4+dx*4.8
!y0=-1.2+dy*2.4
# save the initial data at different time slices
aux xp[1..10]=if((x>0)&(abs(t-2*[j]))<.1)then(x0)else(-100)
aux yp[1..10]=if((x>0)&(abs(t-2*[j]))<.1)then(y0)else(-100)
# set initial data with parameter
glob 0 t {x=x0;y=y0}
# set some options
@ total=22
@ xp=xp5,yp=yp5,xlo=-2.4,ylo=-1.2,xhi=2.4,yhi=1.2,lt=-1
@ maxstor=500000
@ trans=1,dt=2,meth=8
done
```

The lines that start with `!` are treated like invisible parameters that are evaluated anytime the other parameters are changed. Unlike **fixed variables** these are only evaluated once per integration of the equations. The 10 auxiliary variables contain the initial data if they correspond to the time slice and the x coordinate is positive; otherwise they are set to (-100,-100). I use the Dormand-Prince 8(3) integrator as it is very fast especially when the output times are large (e.g., `dt=1`).

Run this file, click on `Initialconds 2 par range` , and fill in the resulting dialog box as:

`Vary1:` dx	`Reset storage:` N
`Start1:` 0	`Use old ic's:` Y
`End1:` 1	`Cycle color:` N
`Vary2:` dy	`Movie:` N
`Start2:` 0	`Crv(1) Array(2):` 2
`End2:` 1	`Steps2:` 101
`Steps:` 101	`:`

and click on **Ok** . A series of small circles will be drawn at different points. Since `xp5,yp5` are in the plot window and the plots are at even times 2, 4, etc, this graph shows all the initial conditions which end up in the right-half plane at $t = 10$. Look at a plot of `xp10,yp10` to see a big difference.

Make a movie. You can make a quick flip-book of the plotted sets at the different time slices as follows. Click on `Viewaxes 2D` and change the plotted variables to `xp1,yp1`, the first in the set. Then click on `Kinescope Capture` to grab the screen image. Next change the view to `xp2,yp2` and capture the screen again with the **K C** commands. Repeat this until you have plotted all 10 sections. Now they are stored in memory. Click on `Kinescope Playback` (**K P**) and click the mouse button to cycle through them. Tap **Esc** when you are tired of doing this.

As an exercise, you could try this method with some other chaotic system like the Lorenz equations or the Rossler attractor.

9.6.4 Poincare maps revisited

The use of adaptive integration methods can make it easy to compute Poincare sections for periodically driven systems. The idea is to just set the output time stem `Dt` to the period of the forcing and then just plot the results. That is, we can ignore the `Poincare` option completely. This only works for systems where the Poincare map is with respect to the time variable. Recall the forced Duffing equation from Chapter 3:

$$x' = v, \qquad v' = x - x^3 - fv + c\cos(\omega t)$$

and the ODE file for it, `duffing.ode`. Run *XPPAUT* with this file. Change `c=.7, f=.3, omega=1.25`. In the `nUmerics` menu, change the `Method` to `DorPri(8)3` with a relative and absolute tolerance of 1e-5. Now click on `Dt` and type the following string `%2*Pi/omega`. *XPPAUT* evaluates command-line numbers that start with the % sign and converts them to floats. Thus, output will only happen at multiples of $2\pi/\omega$, the period. Now change `Total` to `%2*Pi*4000/omega` to get 4000 points! Make a 2D window of (x, v) in the plane, $[-2, 2] \times [-2, 2]$. Finally, edit the graph type (**G E**) and choose `0`. Change the `Linetype` to `-1` for large dots. Now integrate the equations and you will see the usual attractor.

Flagged output Newer versions of *XPPAUT* incorporate a feature for that allows you to make Poincare maps (and many other calculations) very rapidly. The disadvantage of the following method is that you need to add a special line in your ODE file and this cannot be done within the program. Recall from section 3.5 that *XPPAUT* allows you to catch crossings of variables and based on this take some action. One such action is to **output** a point to the data browser. Just include the action, `out_put=1` within the list of actions and when the event occurs, *XPPAUT* will send output to the data browser. Since you generally want to suppress all other output, you should, for example, set `Transient` to some large number from the numerics menu. This guarantees that *XPPAUT* won't print out any other data. The advantage of this over a standard Poincare map is that there can be many possible events that are flagged. It is generally faster than the Poincare map feature since graphic output is suppressed.

9.7 Don't forget...

1. Initial conditions and parameters can be typed in at the command line as formulae. Use the `Parameter` command and then type the name of the parameter. When asked for a value type in the % sign first and then the formula. Use the `Initialconds New` command to type in the initial value in the same manner.

2. There is a stupid little calculator built in to *XPPAUT* . Click on `File Calculator` to get it. You can type in the usual math type expressions to get an answer. Even sums can be evaluated such as `sum(0,100)of(exp(-i'))`. You can set any variable or parameter to an expression by typing, e.g. `x:1/(1+pi^2)` which would set `x` to 0.091999 . . .. Tap `Esc` to exit the calculator.

3. You can run *XPPAUT* over a modem line in `-silent` mode. That is, set up your ODE using the `@` options. Then, type `xpp file.ode -silent` and it will run without ever invoking the interface. The data will be saved in a file `output.dat`. This is also a great way to program in batch mode.

4. Parameter sets are a very useful way to set up problems so that different groups of parameters are associated with nice names that can be invoked by using the `File Get par set` command. For example, consider the 3 different ways to integrate the harmonic oscillator

```
# harmonic oscillator
x'=y
y'=-x
@ dt=.2,total=10
set euler {meth=euler}
set backeul {meth=backeul}
set rk4 {meth=rk4}
done
```

Use it with the `-silent` mode of *XPPAUT* and *XPPAUT* will dump out 3 data files called `euler.dat, backeul.dat, rk4.dat`.

To go one step better, use the parameter sets with the `array` trick to go through a range of sets. Here is a stupid example where I integrate $x' = -x$ for 11 different initial conditions. I use many of the tricks.

```
# integrate over a range of initial conditions
# silently
x'=-x
global 0 t {x=a}
!a=f(index)
par index=0
f(x)=.1*x
set x[0..10] {index=[j]}
@ total=5
done
```

Run this in silent mode and you will get 11 data files that contain the outputs from the solutions to $x' = -x$ with $x(0) = .1 * j$ and $j = 0, \dots, 10$.

Parameter sets can contain any of the declarations used in option files as well as initial data and parameter declarations.

5. Take advantage of the one-parameter range integration as well. If all you are varying is one parameter or initial condition, then you can set the ODE file for a range integration in silent mode. Here are two examples. In the first, one file is produced which contains all the runs:

```
#ex1.ode
x'=y
y'=-x
@ rangeover=y,rangestep=10,rangelow=-1,rangehigh=1
@ range=1,rangereset=no
done
```

Here the initial value of y varies between -1 and 1. If you type `xpp ex1.ode -silent` one output file will be produced that has all the integrations together. There are occasions when you want to do this; I doubt this is such an occasion. In the second example, 11 different files are produced, one for each run:

```
#ex1.ode
x'=y
y'=-x
@ rangeover=y,rangestep=10,rangelow=-1,rangehigh=1
@ range=1,rangereset=yes
done
```

The only difference is `rangereset=yes` which tells *XPPAUT* to reset the storage at every run. In `silent` mode, *XPPAUT* produces files with names, `output.dat.0`, `output.dat.1` , etc.

6. If you have many sets of many parameters and want to run these in a single run in batch mode, keeping each run in a separate file, then here is how to do it. I will use an example with three parameters, a, b, c taking on 4 sets of values, $(1, 2, 3)$, $(1, 2, 4)$, $(-1, 0, 2)$, and $(2, -2, 6)$. I first make a table with 12 entries; the first 3 are for set #1, and so on. Here is the transposed table:

   ```
   12 0 11 1 2 3 1 2 4 -1 0 2 2 -2 6
   ```

   called `pars.tab`. I make an ODE file with a parameter for the run number, use range integration and defined parameters to complete the file. Here is my example:

   ```
   # par.ode
   #
   table pp pars.tab
   par n
   !a=pp(3*n)
   !b=pp(3*n+1)
   !c=pp(3*n+2)
   x'=-x+a
   y'=-y+b
   z'=-z+c
   aux my_a=a
   aux my_b=b
   aux my_c=c
   @ total=10
   @ range=1,rangeover=n,rangelow=0,rangehigh=3,rangestep=3
   @ rangereset=yes
   done
   ```

   I have added auxiliary variables as a "hardcopy" of the actual parameter values used. The `n` parameter runs from 0 to 3 in increments of 1. The table `pp` has a domain of 0 to 11 with the first three integers corresponding to the first three parameter values and so on. That is `pp(2*3+1)` is the value of `b` for the third run. Run this with the line `xpp par.ode -silent` and you will get four files; one for each run.

7. You can make little inline tutorials by using quoted comments that have an action associated with them. The user invokes these by clicking on `File PrtInfo` which creates a window with the ODE source code. Clicking on `Action` in this window produces special comments from the ODE file and certain associated actions. For example here is a linear planar system `planar.ode` with a built in tutorial about the nature of the fixed points:

```
# the linear planar systems
# planar.ode
x'=a*x+b*y
y'=c*x+d*y
par a=-1,b=0,c=0,d=-2
init x=2,y=0
@ xp=x,yp=y,xlo=-5,ylo=-5,xhi=5,yhi=5
# here is a tutorial
"       Linear planar systems
" There are several different behaviors for the phase-plane\
 of a linear 2D system
" To see these click on the (*) and then DirField Flow 5
" {a=-1,b=0,c=0,d=-2} 1. Stable node - two real negative eigenvalues
" {a=0,b=1,c=3,d=0} 2. Saddle point - a positive and negative eigenvalue
" {a=-1,b=3,c=-2,d=0} 3. Stable vortex - complex eigenvalues \
with negative real parts
" {a=.5,b=2,c=-2,d=-.25} 4. Unstable vortex - complex with \
positive real parts
" {a=.5,b=0,c=-.5,d=.5} 5. Unstable node - two positive eigenvalues
" {a=.5,b=2,c=-2,d=-.5} 6. Center - imaginary eigenvalues
" {a=-1,b=1,c=2,d=-2} 7. Zero eigenvalue
"
" Try your own values and classify them!
done
```

Note the lines that have the { } within them will be seen by the user as:

```
* 7. Zero eigenvalue
```

so that what is being done is "invisible." Clicking on the * will result in everything within the curly brackets being done.

9.8 Dynamic linking with external C routines.

XPPAUT has a way to allow you to define the right-hand-sides of your ODE directly in C code. Why would you want to do this? Occasionally, you may have a tremendously complex right-hand-side that is the output from some computer algebra system. Most of these systems, (e.g. Maple or MATHEMATICA) have a command that allows you to output the algebra in C or FORTRAN code. It is then a simple matter to edit this code, compile it, write an ODE file which exploits this and then run *XPPAUT* . Note that you do not have to recompile *XPPAUT* at all. The dynamic linking is done inside *XPPAUT* . You should make sure that you are running a version of *XPPAUT* with dynamic linking enabled. The default is to not use it. Another example is when you just cannot figure out how to implement the right-hand-side in *XPPAUT* .

This example is taken directly out of the users manual for *XPPAUT* . I will write a C file that has three different right-hand-sides. I will then switch between them on the fly while in *XPPAUT* . In order to allow *XPPAUT* to communicate directly with the C file, you have to include a line in the ODE file of the form:

```
export {x,y,a,b,t} {xp,yp}
```

where the first group of variables and parameters are values that you want to pass to the external routine. The second group usually consists of `FIXED` variables that you want the routine to pass back to you. So, here is an ODE file for a two-dimensional system with a bunch of parameters:

```
# tstdll.ode
# test of dll
#
# In XPP click on File-Edit-Load Library
# and choose libexample.so
# Then pick either lv, vdp, duff as the function.
x'=xp
y'=yp
xp=0
yp=0
export {x,y,a,b,c,d,t} {xp,yp}
par a=1,b=1,c=1,d=1
init x=.1,y=.2
done
```

I have exported both the state variables as well as all four parameters and the independent variable, `t`. so, the C program will set the fixed variables, `xp,yp` to the desired values depending on the state variables, etc. These are then the true right-hand sides of the system.

Next, you must write some C code to communicate with *XPPAUT* . Here is the three right-hand example.

```
#include <math.h>
/*
 some example functions
*/

lv(double *in,double *out,int nin,int nout,double *var,double *con)
{
  double x=in[0],y=in[1];
  double a=in[2],b=in[3],c=in[4],d=in[5];
   double t=in[6];
  out[0]=a*x*(b-y);
  out[1]=c*y*(-d+x);
}
```

```
vdp(double *in,double *out,int nin,int nout,double *var,double *con)
{
  double x=in[0],y=in[1];
  double a=in[2],b=in[3],c=in[4],d=in[5];
   double t=in[6];
  out[0]=y;
  out[1]=-x+a*y*(1-x*x);

}

duff(double *in,double *out,int nin,int nout,double *var,double *con)
{
  double x=in[0],y=in[1];
  double a=in[2],b=in[3],c=in[4],d=in[5];
 double t=in[6];
  out[0]=y;
  out[1]=x*(1-x*x)+a*sin(b*t)-c*y;
}
```

Each right-hand-side has the form:

`rhs( double *in, double *out, int nin, int nout, double *var, double *con)`

The double array `in` contains all the exported variables, etc in the order you exported them. So `in[0]` contains the value of `x`. The double array `out` should be set to the values that you want to send back to the fixed variables; `xp` will have the value of `out[0]` etc. The integers, `nin,nout` are just the dimensions of the arrays. Finally, two more arrays are passed which you can generally ignore. These contain the complete list of all state and fixed variables as well all parameters. They are organized as follows for the above example:

```
con[2]=a,con[3]=b,con[4]=c,con[5]=d
v[0]=t,v[1]=x,v[2]=y,v[3]=xp,v[4]=yp
```

That is, the array `con` contains all your parameters in order, starting with `con[2]` and the array `var` contains the time variable, followed by the state variables, and then the fixed variables. Thus, you could directly communicate with all the variables or parameters.

Once you have written your C file, you should compile it and create a library. Here is how to do that:

```
gcc -shared -fpic -o libexample.so funexample.c
```

This just compiles it as a relocatable object file and creates a shared library.

Now run *XPPAUT* with the file `tstdll.ode`. Click on `File Edit Load DLL` . Choose the library you have created (`libexample.so` and then choose one of the following three functions, `lv, vdp, duff` which are respectively the Lotka-Volterra model, the van der Pol oscillator, and the forced Duffing equation respectively. Now integrate the equations and you should see the Lotka-Volterra (or whatever) solutions. Change parameters, etc, and these will be reflected in the solutions to the ODE.

9.8.1 An array example

The `export` directive is useful for small numbers of variables or parameters, but is not so good if you are exporting a big array of values. In this example, I consider an array of 51 coupled oscillators with nearest neighbor coupling. I only need equations for 50 oscillators since the relevant variables are the relative phases. The differential equations are

$$x_j' = H(x_{j+1} - x_j) + H(x_{j-1} - x_j) - x_0'$$

where

$$H(u) = a_0 + a_1 \cos u + a_2 \cos 2u + b_1 \sin u + b_2 \sin 2u + b_3 \sin 3u.$$

I have subtracted off x_0' so that this represents the relative phases. The key point to remember is that the ordering for the storage of the time variable, the dependent variables and the fixed variables is: `t,v1,v2,...,f1,f2,...` where `v` are the variables and `f` are the fixed variables. Here is the ODE file, `chain.ode`:

```
# this is a chain of 50 oscillators using
# dll's to speed up the right-hand sides
x[1..50]'=xp[j]
xp[1..50]=0
par n=50,a0=0,a1=.25,a2=0,b1=1,b2=0,b3=0
export {a0,a1,a2,b1,b2,b3,n}
@ total=100
done
```

Notice that I have defined 50 fixed variables which will contain the right-hand sides. They are set to 0 but will be altered by the external library routines. I pass the number of oscillators and the parameters for the interaction function. I do not need to import anything since this will all be done via the fixed variables. Here is the C code for the right-hand sides:

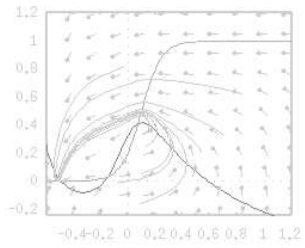
```
#include <math.h>
#define H(u) a0+a1*cos(u)+a2*cos(2*u)+b1*sin(u)+b2*sin(2*u)+b3*sin(3*u)
f(double *in,double *out,int nin,int nout,double *var,double *con)
{
 int i;
 double a0=in[0],a1=in[1],a2=in[2];
 double b1=in[3],b2=in[4],b3=in[5];
 int n=(int)in[6];
 double *x=var+1;
 double *xdot=x+n;
 double x0dot;
 x0dot=H(x[0]);
 xdot[0]=H(-x[0])+H(x[1]-x[0])-x0dot;
 for(i=1;i<(n-1);i++)
 xdot[i]=H(x[i+1]-x[i])+H(x[i-1]-x[i])-x0dot;
 xdot[n-1]=H(x[n-2]-x[n-1])-x0dot;
}
```

This code is very flexible in that I could use it for any size array since the number of oscillators is exported. The time variable, dependent variables, and fixed variables are sent in the array `var`. Thus, `var[0]=t`, `var[1]=x1`, and so on. The fixed variables are `var[1+50]=xp1` etc. These are sent back to XPP to use as the right-hand sides. I define some pointers and some macros to make the file easier to write and read. The pointer `*x=var+1` points to the first variable and `*xp=x+n=var+n+1` points to the first fixed variable.

To run this, fire up *XPPAUT* on the file `chain.ode`. Compile the C file with the line

```
cc -shared -o chain.so -fpic chain.c
```

where I have called it `chain.c`. In *XPPAUT* , click on `File  Edit  Load DLL` . Choose `chain.so` from the list and for the function name, type in `f`. Now let it rip. Change parameters and see what happens.



Appendix A

Colors and linestyles.

XPPAUT uses the numbers from 0-10 to describe colors of plots. These are translated into linestyles for black-and-white postscript files. Here are the color numbers and their approximate names. Figure A shows the corresponding linestyles for black-and-white postscript.

Color 0: Black on a white screen and white on a black screen.

Color 1: Red.

Color 2: Red-orange.

Color 3: Orange.

Color 4: Yellow-orange.

Color 5: Yellow.

Color 6: Yellow-green.

Color 7: Green.

Color 8: Blue-green.

Color 9: Blue.

Color 10: Purple.

Keep these in mind for hardcopy. For example to change the linetype of the nullclines change the color that they are plotted in the program using the options shown in Appendix B and then when hardcopy is made, the linetype will be as shown in the figure. Negative linetypes in *XPPAUT* do not join the points together and instead draw either single points (`LT=0`) or circles or varying diameter, `LT=-1,-2 ,...`

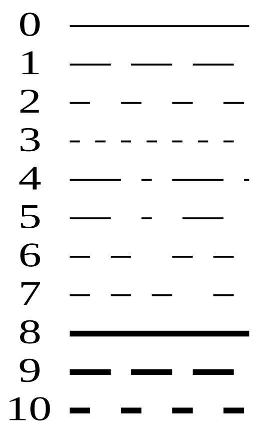
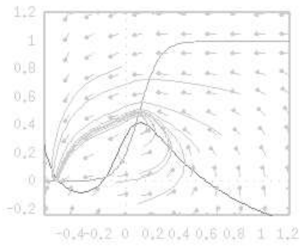


Figure A.1. *Available postscript linestyles.*



Appendix B

The options

XPPAUT enables the user to incorporate a large number of options within an ODE file. These have the form:

```
@ opt1=value,opt2=value2,...
```

where `opt1` is the name of an option and `value` is the assigned value. Almost every option can also be set within *XPPAUT* from the menus. Similarly, most items that you want to set in *XPPAUT* can be set in the ODE file.

The following options *can only be set outside the program.* They are:

- MAXSTOR=`integer` sets the total number of time steps that will be kept in memory. The default is 5000. If you want to perform very long integrations change this to some large number.
- BACK= {`Black,White`} sets the background to black or white.
- SMALL=`fontname` where `fontname` is some font available to your X-server. This sets the "small" font which is used in the Data Browser and in some other windows.
- BIG=`fontname` sets the font for all the menus and popups.
- SMC={0,...,10} sets the stable manifold color
- UMC={0,...,10} sets the unstable manifold color
- XNC={0,...,10} sets the X-nullcline color
- YNC={0,...,10} sets the Y-nullcline color
- OUTPUT=filename sets the filename to which you want to write for "silent" integration. The default is "output.dat".

- BUT=label:sequence. This option can be used repeatedly in the same file and allows you to define a clickable button on the top menu bar of the main window. The `label` defines the label that will appear on the button and the `sequence` is a sequence of keys used for the command. These are the keys you would press to get an item from the main menu (see Appendix E.) For example, if you want to have a one click command for computing the Liapunov exponent, the sequence is `uhl` since you would press `nUmerics stocHastic Liapunov` . Thus, the option is `BUT=liap:uhl`. A quit button would be `BUT=quit:fq` corresponding to `File` . `Quit` .

The remaining options can be set from within the program. They are

Plotting options.

- LT=`int` sets the linetype. It should be less than 2 and greater than -6.
- XP=name sets the name of the variable to plot on the x-axis. The default is `T`, the time-variable.
- YP=name sets the name of the variable on the y-axis.
- ZP=name sets the name of the variable on the z-axis (if the plot is 3D.)
- NPLOT=`int` tells XPP how many plots will be in the opening screen.
- XP2=name,YP2=name,ZP2=name tells XPP the variables on the axes of the second curve; XP8 etc are for the 8th plot. Up to 8 total plots can be specified on opening. They will be given different colors.
- AXES={`2,3`} determine whether a 2D or 3D plot will be displayed.
- PHI=value,THETA=value set the angles for the three-dimensional plots.
- XLO=value,YLO=value,XHI=value,YHI=value set the limits for two-dimensional plots (defaults are 0,-2,20,2 respectively.) Note that for three-dimensional plots, the plot is scaled to a cube with vertices that are ± 1 and this cube is rotated and projected onto the plane so setting these to ± 2 works well for 3D plots.
- XMAX=value, XMIN=value, YMAX=value, YMIN=value, ZMAX=value, ZMIN=value set the scaling for three-d plots.

Numerical options

- SEED=`int` sets the random number generator seed.
- TOTAL=value sets the total amount of time to integrate the equations (default is 20).
- DT=value sets the time step for the integrator (default is 0.05).
- NJMP=`integer` *or* NOUT=`integer` tells *XPPAUT* how frequently to output the solution to the ODE. The default is 1, which means at each integration step. This is also used to specify a the period for maps in the continuation package AUTO.
- T0=value sets the starting time (default is 0).

- TRANS=value tells XPP to integrate until `T=TRANS` and then start plotting solutions (default is 0.)
- NMESH=`integer` sets the mesh size for computing nullclines (default is 40).
- {BANDUP=int, BANDLO=int} sets the upper and lower limits for banded systems which use the banded version of the CVODE integrator.
- METH={ `discrete,euler,modeuler,rungekutta,adams,gear,volterra, backeul, qualrk,stiff,cvode,5dp,83dp,2rb,ymp`} sets the integration method (see below; default is Runge-Kutta.) The latter four are the Dormand-Prince integrators and a Rosenbrock (2,3) integrator and a symplectic integrator.
- DTMIN=value sets the minimum allowable timestep for the Gear integrator.
- DTMAX=value sets the maximum allowable timestep for the Gear integrator
- VMAXPTS=value sets the number of points maintained in for the Volterra integral solver. The default is 4000.
- { JAC_EPS=value, NEWT_TOL=value, NEWT_ITER=value} set parameters for the root finders.
- ATOLER=value sets the absolute tolerance for several of the integrators.
- TOLER=value sets the error tolerance for the Gear, adaptive RK, and stiff integrators. It is the relative tolerance for CVODE and the Dormand-Prince integrators.
- BOUND=value sets the maximum bound any plotted variable can reach in magnitude. If any plottable quantity exceeds this, the integrator will halt with a warning. The program will not stop however (default is 100.)
- DELAY=value sets the maximum delay allowed in the integration (default is 0.)
- AUTOEVAL={`0,1`} tells XPP whether or not to automatically re-evaluate tables every time a parameter is changed. The default is to do this. However for random tables, you may want this off. Each table can be flagged individually within XPP.

Poincare map

- POIMAP={ `section,maxmin,period`} sets up a Poincare map for either sections of a variable, the extrema, or period between events.
- POIVAR=name sets the variable name whose section you are interested in finding.
- POIPLN=value is the value of the section; it is a floating point.
- POISGN={ `1, -1, 0` } determines the direction of the section.
- POISTOP=1 means to stop the integration when the section is reached.

Range integration

- RANGE=1 means that you want to run a range integration (in batch mode).
- RANGEOVER=name, RANGESTEP, RANGELOW, RANGEHIGH, RANGERESET=`Yes,No`, RANGEOLDIC=`Yes,No` all correspond to the entries in the range integration option.

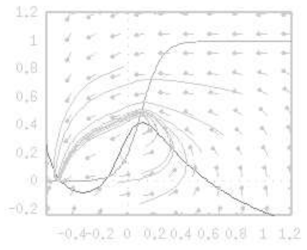
Phasespace

- TOR_PER=value, defined the period for a toroidal phasespace and tells XPP that there will be some variables on the circle.
- FOLD=name, tells XPP that the variable ¡name¿ is to be considered modulo the period. You can repeat this for many variables.

AUTO options. The following AUTO-specific variables can also be set: `NTST, NMAX, NPR, DSMIN, DSMAX, DS, EPSS,EPSL,EPSU, PARMIN,PARMAX, NORMMIN, NORMMAX, AUTOXMIN, AUTOXMAX, AUTOYMIN, AUTOYMAX, AUTOVAR`. The last is the variable to plot on the y-axis. The x-axis variable is always the first parameter in the ODE file unless you change it within AUTO.

Miscellaneous

- AUTOEVAL={`1,0`} tells *XPPAUT* whether or not to recompute tables whenever parameters are changed. If you have a table of random connectivity, you want this turned off (0).
- BELL=0 turns of the bell for *XPPAUT* .
- COLORMAP={`0,1,2,3,4,5`} switches the color map:
  - 0: standard
  - 1: periodic
  - 2: "hot"
  - 3: "cool"
  - 4: red-blue
  - 5: grey



Appendix C

Numerical methods

Here I briefly describe the numerical methods used in *XPPAUT* .

C.1 Fixed points and stability

XPPAUT can find fixed points of maps and differential equations. Finding fixed points involves solving:

$$G(X) = 0$$

where $G(X) = F(X)$ for differential equations, $X' = F(X)$ and $G(X) = X - F(X)$ for maps, $X_{n+1} = F(X_n)$. Newton's method is iterative and satisfies the scheme:

$$X_{k+1} = X_k - J^{-1}G(X_k)$$

where J is the matrix of partial derivatives of G evaluated at X_k. *XPPAUT* 's implementation of Newton's method requires three parameters: the maximum number of iterates, the tolerance, and a parameter for the numerical computation of the matrix, J called `epsilon` in *XPPAUT* . If the difference between successive iterates falls within the tolerance or $G(X_k)$ falls within the tolerance, then convergence is assumed and the root is found. The matrix J is found by perturbing each of the variables by an amount proportional to `epsilon` and then using this perturbation to approximate the derivative.

Once a fixed point is found the matrix J is evaluated one last time and the eigenvalues of J are computed using standard linear algebra routines. (See *Numerical Recipes* for example.) *XPPAUT* uses this information to compute the stability of solutions. One-dimensional stable and unstable invariant manifolds are computed by finding the eigenvector corresponding to the real eigenvalue. A small perturbation from the fixed point along this vector is made and the equations are integrated.

The eigenvector is computed by inverse iteration. That is, suppose λ is an eigenvalue. Let $M = J - \nu I$ where ν is a number close to λ. *XPPAUT* chooses a random unit vector and multiplies it by M^{-1} and normalizes the result to get a new vector. This iterative scheme continues until convergence is achieved. The resulting vector is an approximate eigenvector.

C.2 Integrators.

The numerical solutions of ordinary differential equations is an old and well-studied field. *XPPAUT* has essentially two classes of integrators: (i) fixed step size and (ii) adaptive step size. These can each be divided further into two classes: (i) explicit and (ii) implicit. This latter distinction is best illustrated by considering the two simplest fixed step integrators, Euler and backward Euler. Euler's method solves

$$\frac{dx}{dt} = f(x,t)$$

by the approximation

$$x(t+h) = x(t) + hf(x(t),t),$$

where h is the step size. This is the easiest method to code and to understand. However, it suffers from the two major trouble spots of numerical integration: accuracy and stability. Any numerical analysis book will tell you that the accuracy is only $O(h)$. That is, let T be a fixed interval of time. Let n be defined by $nh = T$. Then

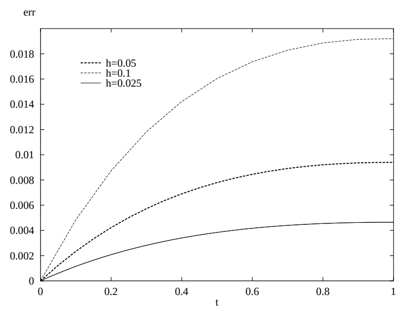
$$|x(T) - x_n| = O(h),$$

where x_n is the n^{th} iterate of Euler's method. Thus, to halve the error, you must halve the step size. This problem can be somewhat rectified by using higher order methods. The best known of these are the Runge-Kutta (RK) methods which make several calls to the right-hand side. For example, *XPPAUT* uses Heun's or modified/improved Euler method, a second order RK method defined by:

$$\begin{aligned} y_1 &= x_n + hf(x_n,t) \\ y_2 &= x_n + hf(y_1,t+h) \\ x_{n+1} &= \frac{1}{2}(y_1 + y_2). \end{aligned}$$

Note that this requires two evaluations of the right-hand sides but gives $O(h^2)$ accuracy. The advantage of this is that halving the step size quadruples the accuracy! Figure C.2 illustrates this point. Try to reproduce this figure with the following ODE file:

```
# a first order equation whose exact answer is known
x'=1-x
aux 1-exp(-t)
aux err=abs(1-exp(-t)-x)
@ total=2
done
```



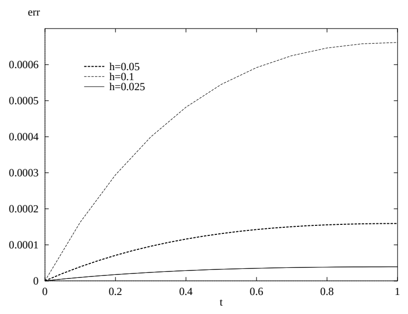


Figure C.1. *Error from Euler (left) and improved Euler (right).*

The classical Runge-Kutta method is a four-step method that gives $O(h^4)$ accuracy so that halving the step size results in a 16-fold decrease in error:

$$\begin{aligned} k_1 &= f(t, x_n) \\ k_2 &= f(t + h/2, x_n + hk_1/2) \\ k_3 &= f(t + h/2, x_n + hk_2/2) \\ k_4 &= f(t + h, x_n + hk_3) \\ x_{n+1} &= x_n + \frac{h}{6}(k_1 + 2k_2 + 2k_3 + k_4). \end{aligned}$$

These methods are called explicit since the evaluation of the next term depends *only* on the previous values. Explicit methods all suffer from *stability* problems. This is nicely illustrated by the following nonlinear equation:

$$x' = ax(1 - x) \qquad x(0) = 0.1. \tag{C.1}$$

For all $a > 0$ solutions to this converge monotonically to $x = 1$. Consider Euler's method applied to this:

$$x_{n+1} = x_n + hax_n(1 - x_n) = hax_n(1 + \frac{1}{ha} - x_n).$$

The latter will (I hope) be recognized as a variant of the famous logistic map which as all sorts of chaotic behavior. That is, fix the step size and let the parameter a get larger. Then clearly, once a goes beyond about $2/h$, we can expect to see the fixed point, $x_n = 1$ become unstable. Integrate (C.1) over the interval [0,1] with a step-size of 0.05 using Euler's method for the following values of a: (20,30,40,60,80). Even the classic Runge-Kutta goes bad. Try using Runge-Kutta with the default stepsize and $a = 90, 100, 101, 102$.

The solution to this problem is to use a *backward* or *implicit* method. Consider the simple linear differential equation:

$$x' = -ax \qquad x(0) = 1$$

where a is a parameter. The Euler iteration with stepsize, h, for this is $x_{n+1} = (1-ah)x_n$ with a solution, $x_n = (1-ah)^n$. Clearly, if $ah > 1$ then the solution is not monotonic but instead undergoes oscillations. If $ah > 2$ then solutions grow in an oscillatory manner. We saw this behavior in the nonlinear logistic model. Implicit methods avoid some of these problems. The simplest method is called implicit Euler and the scheme is:

$$x_{n+1} = x_n + hf(t+h, x_{n+1}).$$

The big difference between this and Euler is that the right-hand side depends on x_{n+1} so that at each time step, we have to solve a (nonlinear) equation for x_{n+1}. However, the advantage of this method becomes obvious if we apply it to the linear equation above:

$$x_{n+1} = x_n - hx_{n+1}$$

and solve for x_{n+1} to get $x_{n+1} = x_n/(1+ha)$. Thus, the solution is

$$x_n = x_0 \left(\frac{1}{1+ah} \right)^n$$

which is monotonically decreasing. Thus, unlike the explicit Euler method, backward Euler behaves like the differential equation no matter how large a is. This idea can be applied to any linear system and the result is

$$x_{n+1} = (I - hA)^{-1} x_n.$$

If all the eigenvalues of A have negative real parts, then this iteration scheme always contracts initial data to the origin for any positive value of h. A numerical method is stable if it has the same property as backward Euler. *XPPAUT* implements backward Euler for the general nonlinear equation. This means that at each step it must solve the nonlinear equation:

$$x_{n+1} = x_n + hF(t+h, x_{n+1}).$$

Newton's method is used to solve this. *XPPAUT* requires two parameters, `MaxIt` the maximum number of iterates and `Tolerance` the tolerance for convergence. (See above for a description of Newton's method.)

XPPAUT has one more standard fixed step integrator: Adams-Bashforth which is a fourth-order predictor corrector. I will not describe this method except to say that it requires fewer function evaluations than Runge-Kutta while maintaining fourth order accuracy. It is an explicit method and quite unstable. In fact, try Adams on the logistic equation and figure out at which value of a the method becomes unreliable with a time step of 0.05. I don't recommend this method as it is quite unstable and the extra speed up you get over Runge-Kutta is not worth the risk.

XPPAUT also has a symplectic integrator. These integrators should only be used on systems of the form:

$$\ddot{x_j} = f_j(x_1, \ldots, x_n)$$

written as follows:

$$\dot{x_j} = v_j, \qquad \dot{v_j} = f_j.$$

Symplectic integrators obey the discrete equivalent of a conservation law so that they will preserve energy for Hamiltonian systems. They should not be used for other types of systems.

In addition to the fixed step integrators, *XPPAUT* has several adaptive integrators. Four of them, Stiff, Gear, Rosenbrock and Cvode are implicit and work well for stiff differential equations. Stiff is based on the stiff algorithm in Numerical Recipes in C; Gear is a direct translation of FORTRAN code found in Gear's book; CVODE is based on LSODE and was obtained at the web site `http://netlib.bell-labs.com/netlib/ode/index.html`; and Rosenbrock is based on an algorithm called `ODE23S` found in The other three adaptive integrators: Dormand-Prince 8(3), Dormand-Prince 5, and QualRk are all explicit. DorPri8(3) is based on a Runge-Kutta method of order 8 with automatic step size control. It was developed by Prince and Dormand and is described in Hairer (1993). It provides a piecewise polynomial approximation of seventh order to the solution. DorPri(5) is similar and of lower order. Both methods are complicated and the reference to Hairer should be consulted for details. Both require a relative and an absolute tolerance. Relative tolerance is scaled by the magnitude of the output while absolute tolerance is not. If problems occur during the integration, *XPPAUT* puts out error messages that "suggest" possible remedies. QualRk is an adaptive-step fourth order Runge-Kutta method. It is based on code in Numerical Recipes. The interested user should consult that reference. However, I point out that the basic idea is to take a Runge-Kutta step of size h and then two steps of size $h/2$ and compare the results. The difference gives a good estimate of the error. This is used to adjust the time step. The advantage of adaptive techniques is that it may be possible to take big steps sometimes and switch to small steps only when necessary so that time is not wasted.

C.2.1 Delay Equations

Delay equations are solved by using cubic polynomial interpolation to obtain the delayed value. *XPPAUT* stores output from the integrators on a circular stack at regular output intervals. The size of the stack is determined by the parameter `Maximum Delay`. When $u(t_0)$ is needed for some value of t_0, the values of u at nearby points are used to interpolate the new value. If the delay is a multiple of the output times, no interpolation is needed. Thus, if you are not using the delay as a parameter, it is most accurate to choose `Delta T` accordingly.

Stability of fixed points for delay equations is done as follows. The linearization results in a system of the form:

$$\frac{dy}{dt} = A_0 y(t) + \sum_{j=1}^{m} A_j y(t - \tau_j)$$

where A_j are matrices. Substitution of $y(t) = \Phi \exp(\lambda t)$ yields the transcendental characteristic equation:

$$\Delta(\lambda) = \det(A_0 + \sum_{j=1}^{m} A_j e^{-\lambda \tau_j} - \lambda I).$$

We are interested only in positive eigenvalues. Bounds on the largest positive real part are readily obtained. *XPPAUT* then computes an approximate contour integral of the function $\Delta(\lambda)$ in a big square in the right-half plane. From the argument principle, we can thus obtain the number of roots in the contour and thus the stability of the fixed point. *XPPAUT* then uses Newton's method to find the root of $\Delta(\lambda)$ closest to 0.

C.2.2 The Volterra Integrator

This is based on a method in Peter Linz' book on solving integral equations. It is an implicit first order method. I will show how it works for the scalar integral equation:

$$u(t) = f(t) + \int_0^t F(t, s, u(s))\, ds.$$

Let $t = hn$ where h is the time-step. Then, we can write the discretization of this as:

$$u_n = f_n + h \sum_{j=0}^{n-1} F(nh, nj, u_j) + hF(nh, nh, u_n).$$

Note that u_n appears on the right-hand side. Thus, we have to once again use Newton's method to find u_n. This method is stable for the same reason that backward Euler is stable.

Consider the singular integral:

$$\int_0^{nh} \frac{F(nh, s, u(s))}{(nh - s)^r}\, ds$$

where $0 < r < 1$. Then,

$$\begin{aligned}\int_0^{nh} \frac{F(nh, s, u(s))}{(nh - s)^r}\, ds &\approx \sum_{j=0}^{n-1} F(nh, nj, u(nj)) \int_{jh}^{(j+1)h} \frac{ds}{(nh - s)^r} \\ &= \frac{1}{1-r} \sum_{j=0}^{n-1} F(nh, nj, u(nj)) \left([nh - jh]^{1-r} - [nh - (j+1)h]^{1-r} \right).\end{aligned}$$

This trick changes the singular integral into a regular sum.

C.3 How AUTO works.

AUTO is capable of finding fixed points of systems and tracking them as a parameter varies as well as tracking solutions to differential equations. In the latter case, the problem is always regarded as a boundary value problem and as such is always solved on the unit interval. Consider first, the solutions to algebraic problems:

$$f(u,\lambda)=0, \quad u,f\in R^n, \quad \lambda\in R.$$

This is done by parameterizing the solutions by a parameter s so that solutions of the form $(u(s),\lambda(s))$ must be found. Differentiate the equation $f(u(s),\lambda(s))=0$ with respect to s and we obtain an $n\times n+1$ dimensional matrix

$$M=(f_u|f_\lambda)$$

which is assumed to have a one-dimensional nullspace. The nullvector, $(\dot{u},\dot{\lambda})$ is normalized as

$$c_u\dot{u}^T\dot{u}+c_\lambda\dot{\lambda}^2=1$$

where c_u, c_λ are constants. Suppose that we have already found (u_{j-1},λ_{j-1}) and $(\dot{u}_{j-1},\dot{\lambda}_{j-1})$. Then we must solve

$$\begin{aligned} f(u_j,v_j)&=0\\ \Delta s&=c_u(u_j-u_{j-1})^T\dot{u}_{j-1}+c_\lambda(\lambda_j-\lambda_{j-1})\dot{\lambda}_{j-1}\end{aligned}$$

where Δs is the desired step size. This is now an $(n+1)\times(n+1)$ system and the Jacobian matrix is

$$M\equiv\begin{pmatrix} f_u & f_\lambda\\ \dot{u} & \dot{\lambda}\end{pmatrix}$$

which is not singular even if f_u is. Thus, AUTO can track around fold points. Limit or fold points are tracked with two parameters (λ,μ) by applying the same techniques to the $2n+1$-dimensional system

$$f(u,\lambda,\mu)=0, \quad f_u(u,\lambda,\mu)v=0, \quad v^Tv-1=0.$$

Continuation of Hopf bifurcation points is a bit more complicated but depends on the fact that at a Hopf bifurcation, the linearized problem

$$V(t)'=Tf_uV(T)$$

will have a nonconstant solution satisfying $V(0)=V(1)=0$ for some value of T. Since solutions will be proportional to sine and cosine at the Hopf point, $V(t)=\xi\sin 2\pi t+\eta\cos 2\pi t$. The result is the two algebraic equations

$$(T/2\pi)f_u\xi+\eta=0, \quad -\xi+(T/2\pi)f_u\eta=0,$$

and the normalization

$$\xi^T\xi+\eta^T\eta=1.$$

Due to translation invariance, one more normalization condition must be provided to fix a phase:

$$\eta_0\xi-\xi_0\eta=0$$

where ξ_0, η_0 are the previously found solutions. Thus we now have a $3n+2$ dimensional system to solve. Continuation for maps is done in a similar manner. For differential equations on the interval $[0,1]$, AUTO solves an algebraic system that is obtained by intelligently discretizing the ODE and then solving the resulting algebraic equations. The parameter `Ntst` in the AUTO Numerics menu defines the number of points in the discretization. The discretization is not uniform, rather, more points are placed where the solutions is changing rapidly. Thus, there is never really any integration of solutions of the equations in AUTO, instead, continuation of systems of algebraic equations is done.

Here is a detailed example of how AUTO continues equations. Consider

$$f(u,\lambda) = \lambda + u^2 = 0.$$

Suppose we start at $\lambda = -1$ and $u = 1$. The normalized nullvector for $f_u \dot{u} + \dot{\lambda} = 0$ is given by

$$(\dot{u}, \dot{\lambda}) = \frac{1}{1+4u^2}(-1, 2u).$$

where we use the fact that $f_u = 2u$. We must solve:

$$\begin{aligned} u_j^2 + \lambda_j &= 0 \\ \Delta s &= \frac{1}{1+4u_{j-1}^2}[(u_j - u_{j-1})(-1) + (\lambda_j - \lambda_{j-1})2u_{j-1}] \end{aligned}$$

and the Jacobian of this pair of equations at the guessed value, u is

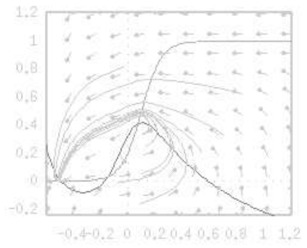
$$M = \frac{1}{1+4u_{j-1}^2}\begin{pmatrix} 2u & 1 \\ -1 & 2u_{j-1} \end{pmatrix}.$$

The determinant of this matrix is

$$\frac{1+4uu_{j-1}}{1+4u_{j-1}^2}$$

which will be nonzero if we move a small enough step so that u and u_{j-1} are close.

More details on AUTO and the methods used can be found in the references.



Appendix D

Structure of ODE files.

All ODE files are just text files which consist of a series of definitions and directives to the *XPPAUT* parser. The order in which the definitions are given is usually unimportant with one **major exception**. So-called fixed or temporary variables are evaluated in the order in which they are defined. Thus, you should **never** use a named fix variable before it is defined as this will lead to some rather bizarre results when you attempt to solve an ODE. ODE files are all line oriented with a limit of 1024 characters per line. You can use the standard line continuation symbol \. Here are all the possible ODE file directives:

#: comment line. This is ignored by *XPPAUT*

": A comment that is extracted and can be displayed in a separate window.

" {a=1,b=2,...}: A comment with some "action" associated with it. When these comments are displayed in the comment window, an * appears next to them. When clicked by the user, various initial conditions, parameters, and XPP internal options can be set.

`!name=formula` This defines a *derived* parameter, `name` whose values could depend on other parameters. These do not appear in the parameter list since they are presumably tied to the other parameters. Each time you change a parameter, they also are updated.

options `filename`: Insert a file name with common options. This is include only for backward compatibility and is generally obsolete.

name(t+1)=`formula` or

dname/dt=`formula` or

name'=`formula`: Define a differential/difference equation dependent variable, **name** and the `formula` defining its right-hand-side. For example

```
x'=-x+sin(t)
z(t+1)=z*(4-z)
dw/dt=1-cos(w)+(1+cos(w))*a
```

name(t)=`formula` or

volterra `name = formula`: Define a Volterra integral equation for the variable **name**. For example

```
y(t)=int{exp(-t)#x}
y'=-y + a int{exp(-(t-t'))*max(y-1/(1+(t-t')^2),0)}
y(t)=int[.5]{1#y}
```

The # symbol means a convolution and `[a]` multiplies the integrand by $1/(t-t')^a$ where $0 < a < 1$.

markov `name nstates`

$$\begin{array}{llll}
\{P_{0,0}\} & \{P_{0,1}\} & \dots & \{P_{0,n-1}\} \\
\{P_{1,0}\} & \{P_{1,1}\} & \dots & \{P_{1,n-1}\} \\
\vdots & \vdots & \dots & \vdots \\
\{P_{n-1,0}\} & \{P_{n-1,1}\} & \dots & \{P_{n-1,n-1}\}
\end{array}$$

This defines a Markov variable, **name** which has `nstates` states. Then following is the transition table as a series of formulas that are delimited by the curly brackets {}. The diagonal entries are ignored but should still contain a number or formula. For example:

```
markov z 2
{0} {alpha(v)}
{beta(v)} {0}
```

aux name=formula defines a named quantity, **name** which appears in the data browser and is available for plotting. Note that **auxiliary** variables are not known internally to *XPPAUT* so that you can't use them in formulas

name= `formula` defines an internal or quantity which can be used in other formulas. This `fixed` variable serves the same function as temporary quantities in C code.

par `name1=value1, name2=value2, ...` defines named parameters with default values. These can later be altered and varied in *XPPAUT* . **Never put spaces between the name, value, and equal sign**. For example:

```
par a=1,b=2,c=2,big_g=20
```

number `name1=value1, name2=value2, ...` defines named parameters with default values. These are fixed and cannot be changed in *XPPAUT* . For example:

```
number faraday=96485,rgas=8.3147,tabs0=273.15
```

name(x,y,...)=`f(x,y,...)` defines a function of up to 9 variables. This function can be used in any right-hand side or other functions. For example:

```
f(x)=if(x>1)then(1/ln(x/(x-1)))else(0)
g(v,w)=2v^2(1-w)-w/10
```

table `name filename` defines a lookup table called **name** defined the values found in the file, **filename**. This behaves as a one variable function which uses linear interpolation on the values of the tabulated file. The file has the structure:

```
npts
xlo
xhi
y(xlo)
...
y(xhi)
```

The domain of the function is $[x_{lo}, x_{hi}]$.

table `% name npts xlo xhi f(t)` defines a precomputed table called **name** using **npts** and evaluating **f(t)** at $t = x_{lo}, \ldots, x_{hi}$. For example:

```
table h % 101 0 6.283 sin(t)+.6+.4*cos(t)+.2*sin(2*t)
```

Tables are automatically re-evaluated whenever you change parameters unless you turn off the AUTOEVALUATE flag.

wiener `name1, name2, ...` defines a series of normally distributed random variables with mean 0 and standard deviation $\sqrt{dt}$ where dt is the internal time step. The advantage of using **wiener** is that when you change the time-step, the magnitude of these variables is automatically adjusted.

global `sign condition {name1=form1;...}` defines a global flag allowing *XPPAUT* to implement delta functions. If **sign=1** and **condition** changes from negative to positive or if **sign=-1** and **condition** changes from positive to negative, or if **sign=0** and **condition** vanishes identically, then each of the variables, **name** is immediately changed to the value of **formula**. For example:

```
global 1 x-1 {x=0;y=y+a*sin(y)}
```

init `name1=val1,name2=val2,...` sets the initial data of **name** to the number **val**.

name(0)= `expression` sets the initial value of **name** to the expression. If **name** is a delay-differential equation variable, then if **expression** is a function of time, **t**, then **name(t)** is set to the function for $-M < t < 0$ where M is the maximum delay. For example:

```
x'=-delay(x,2)
x(0)=sin(t)
```

bdry `expression` defines a boundary condition at the start or the end of an interval. If the variable name is primed, then it refers to the end of the interval, otherwise to the start of the interval. For example:

```
bdry u'-v-1
```

forces the boundary condition $u(L) - v(0) - 1 = 0$ where $[0, L]$ is the interval for the ODE.

0= `expression` defines an algebraic condition for differential-algebraic equations.

solve `name=expression` tells *XPPAUT* to solve the algebraic conditions for the variable **name** with initial guess **expression**. For example:

```
x'=-y
init x=0
0=y+exp(y)-x
solv y=-.56715
```

Solves the DAE $x' = -y$ where $y + e^y = x$. Note that there is no closed form inverse of $y + e^y$.

special `name=FUN(...)` defines a one-dimensional array **name** that represents a special summation involving an array of variables and a table. **FUN** can be one of the following:

conv(type,n,m,w,u):

$$\text{name}(j) = \sum_{l=-m}^{m} w(m+l)\text{shift}(u, j+l) \qquad j = 0, \ldots n-1$$

where w is a table of length $2m + 1$ and **type={even,0,periodic}** determines how $u(j + l)$ is defined for $j + l < 0$ and $j + l \geq n$.

fconv(type,n,m,w,u,v,f):

$$\text{name}(j) = \sum_{l=-m}^{m} w(m+l)f(u(j+l), v(j)) \qquad j = 0, \ldots n-1$$

sparse(n,m,w,l,u) evaluates to

$$\text{name}(j) = \sum_{l=0}^{m-1} w(jm+l)u(c(jm+l)) \qquad j = 0, \ldots, n-1$$

where w is a table of length mn and c is a table of indices of length mn.

fsparse(n,m,w,l,u,v,f) evaluates to

$$\text{name}(j) = \sum_{l=0}^{m-1} w(jm+l)f(u(c(im+l)), v(j))$$

mmult(m,n,w,u) evaluates to:

$$\text{name}(j) = \sum_{l=0}^{m-1} w(l+mj)u(l) \qquad j = 0, \ldots, n-1$$

fmmult(m,n,w,u,v,f) evaluates to:

$$\text{name}(j) = \sum_{l=0}^{m-1} w(l+mj)f(u(l), v(j)) \qquad j = 0, \ldots, n-1$$

These can then be used in right-hand sides of ODEs. For example

```
table w % 21 -10 10 exp(-abs(t))
special k=conv(even,101,21,w,u0)
u[0..100]'=-u[j]+f(k([j]))
```

Here `k` evaluates as the discrete convolution of $\exp(-|x|)$ with the array u.

set `name` {`x1=z1,x2=z2,...,`} defines a named set of declarations including parameter values, initial data, and options, which can be invoked while running *XPPAUT* with the `File Get par set` command. For example:

```
set hopf {a=.3,x=1.2,b=9}
```

@ `opt1=val1,opt2=val2,...` sets various internal *XPPAUT* options. For example:

```
@ maxstor=100000,total=1000,dt=.02
```

anything[j1..j2] `expr[j]` is expanded by *XPPAUT* into a series of $j_2 - j_1 + 1$ statements e.g.

```
x[1..4]'=-x[j]+[j]
```

is expanded into:

```
x1'=-x1+1
x2'=-x2+2
x3'=-x3+3
x4'=-x4+4
```

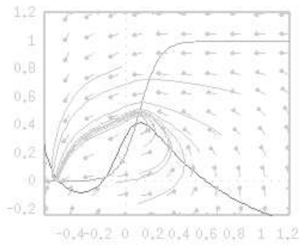
%[j1..j2] expands all following statements until another **%** is encountered as above. For example:

```
%[1..4]
x[j]'=-y[j]-x[j-1]
y[j]'=x[j]
%
```

is expanded into:

```
x1'=-y1-x0
y1'=x1
x2'=-y2-x1
y2'=x2
x3'=-y3-x2
y3'=x3
x4'=-y4-x3
y4'=x4
```

done tells *XPPAUT* the file is complete.



Appendix E

Complete command list

Here we lay out the tree structure of all the commands. They are not described, however. All keyboard shortcuts are capitalized and indicated in boldface. Thus, to get the command to fit data, you could click **U H I** or choose `nUmerics stocHastic fIt data` .

E.1 Main menu

- **I**nitialconds: **R**ange, **2** par ranges, **L**ast, **O**ld, **G**o, **M**ouse, **S**hift, **N**ew, s**H**oot, **F**ile, form**U**la, m**I**ce, **D**AE guess, **B**ackward,
- **C**ontinue
- **N**ullcline: **N**ew, **R**estore, **A**uto, **M**anual,
  - **F**reeze: **F**reeze, **D**elete all, **R**ange, **A**nimate
- **D**ir field/flow: **D**irect Field, **F**low, **N**o dir fld, **C**olorize, **S**caled Dir Fld.
- **W**indow/zoom: **W**indow, **Z**oom In, Zoom **O**ut, **F**it
- pha**A**sespace: **A**ll, **N**one, **C**hoose
- **K**inescope: **C**apture, **R**eset, **P**layback, **A**utoplay, **S**ave, **M**ake Anigif
- **G**raphic stuff: **A**dd curve, **D**elete last, **R**emove all, **E**dit curve, **P**ostscript, a**X**es opts, exp**O**rt, **C**olormap,
  - **F**reeze: **F**reeze, **D**elete, **E**dit, **R**emove all, **K**ey, **B**if Diag, **C**lr BD, **O**n/ff freeze
- n**U**merics: **T**otal, **S**tart time, t**R**ansient, **D**t, **N**cline ctrl, s**I**ng pt ctrl, n**O**utput, **B**ounds, **M**ethod, d**E**lay, r**U**elle plot, looK**u**p, bnd**V**al, **ESC**-exit,

  - stoc**H**astic: **N**ew seed, **C**ompute, **D**ata, **M**ean, **V**ariance, **H**istogram, **O**ld hist, **F**ourier, **P**ower, f**I**t data, **L**iapunov, **S**tat
  - **P**oincare map: **N**one, **S**ection, **M**ax/min, **P**eriod
  - **A**veraging: **N**ew adjoint, **M**ake H, **A**djoint, **O**rbit, **H**fun

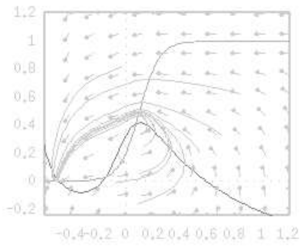
- **F**ile: **P**rt info, **W**rite set,**R**ead set, **A**uto, **C**alculator, **S**ave info, **B**ell off, c-**H**ints, **Q**uit, **T**ranspose, t**I**ps, **G**et par set
  - **E**dit: **R**HS's,**F**unctions, **S**ave as, **L**oad DLL
- **P**arameters
- **E**rase
- **M**ake window: **C**reate, **K**ill all, **D**estroy, **B**ottom, **A**uto, **M**anual, **S**imPlot On/Off
- **T**ext etc: **T**ext, **A**rrow, **P**ointer, **M**arker, **D**elete all, marker**S**.
  - **E**dit: **C**hange, **M**ove, **D**elete
- **S**ing pts: **G**o,**M**ouse, **R**ange, monte**C**arlo
- **V**iew axes: **2**D, **3**D, **A**rray, **T**oon
- **X**i vs T
- **R**estore
- **3**D-params
- **B**ndryval: **R**ange, **N**o show, **S**how, **P**eriodic.

E.2 AUTO

- **P**arameter
- **A**xes: **H**i, **N**orm, h**I**-lo, **P**eriod, **T**wo par, **Z**oom, last **1** par, last **2** par, **F**it, f**R**equency, **A**verage
- **N**umerics
- **R**un
- **G**rab
- **U**sr period
- **C**lear
- re**D**raw
- **F**ile: **I**mport orbit, **S**ave diagram, **L**oad diagram, **P**ostscript, **R**eset diagram, **C**lear grab, **W**rite pts, **A**ll info

E.3 Browser commands

- **F**ind
- **G**et
- **R**eplace
- **L**oad
- **W**rite
- **U**nreplace
- **T**able
- **A**dd col
- **D**el col



Appendix F

Error messages.

XPPAUT is full of obscure error messages. Here I provide a list of common ones, where they come from, and what to do about them.

Out of bounds Some variable or auxiliary quantity went beyond the proscribed maximum. Fix this by increasing the `Bounds` in the `nUmerics` menu. But be careful, some solutions to ODEs really do blow up.

Bad formula The parser was unable to figure out what you meant. You have probably mistyped a name, forgotten a parenthesis, or used a name not known to *XPPAUT* .

Singular Jacobian In Newton's method, the matrix was not invertible. In boundary value problems, this sometimes means that you have not written the boundary conditions correctly. In integration, sometimes setting the time step to be smaller works.

Maximum iterates exceeded Again in newtons method or in curve fitting, the specified tolerances could not be achieved in the given number of iterates. Either loosen tolerances or increase the maximal number of iterates.

Delay negative or too large. Increase the maximal allowable delay. If the delay is negative, you have an ill posed problem.

Could not converge to root. In solving a nonlinear equation, the Newton solver failed to converge. You should make a better guess.

Working too hard??? This error occurs when you have global flags. It is a difficult error to fix. It generally means that when condition A occurs it induces condition B which in turn induces condition A again and so on. Hitting `Escape` many times will sometimes work, but occasionally you need to kill *XPPAUT* .

All curves used. When you freeze a plot, there are only 26 per graph window. Delete some.

Bad condition. Ignoring occurs in the computation of histograms. If the formula for the condition is bad, *XPPAUT* just ignores it.

Out of memory. Time to upgrade!

Out of film. In the Kinescope command, only so many screen shots can be saved.

No prior solution. You tried the `Initialconds Last` command without first computing a solution with new initial data.

No shooting data available. You have to find a fixed point to your system with a one-dimensional invariant manifold before you can use this.

Incompatible parameters. You tried to load a saved set file that is not associated with the ODE file you are running.

F.1 Cheat sheet

FILE FORMAT

```
# comments
d<name>/dt=<formula>
<name>'=<formula>
<name>(t)=<formula>
volt <name>=<formula>
<name>(t+1)=<formula>
# arrays
x[n1..n2]' = ...[j] [j-1]
markov <name> <nstates>
  {t01} {t02} ... {t0k-1}
  {t10} ...
  ...
  {tk-1,0} ... {tk-1 k-1}

aux <name>=<formula>
# parameters defined as formulae:
!<name>=<formula>
<name>=<formula>
parameter <name1>=<value1>,<name2>=<value2>, ...
wiener <name1>, <name2>, ...
number <name1>=<value1>,<name2>=<value2>, ...
<name>(<x1>,<x2>,...,<xn>)=<formula>
table <name> <filename>
table <name> % <npts> <xlo> <xhi> <function(t)>
global sign {condition} {name1=form1;...}
init <name>=<value>,...
# delay initial conditions:
<name>(0)=<value> or <expr>
bdry <expression>
0= <expression>    <---  For DAEs
solv <name>=<expression> <------ For DAEs
special <name>=conv(type,npts,ncon,
                    wgt,rootname)
  fconv(type,npts,ncon,wgt,rootname,
        root2,function)
  sparse(npts,ncon,wgt,
         index,rootname)
  fsparse(npts,ncon,wgt,index,rootname,
          root2,function)
# comments
@ <name>=<value>, ...
set <name> {x1=z1,x2=z2,...}
options <filename>
```

```
# Active comments
" {z=3,b=3,...} Some nice text
done
```

INTEGRAL EQUATIONS

The general integral equation

$$u(t) = f(t) + \int_0^t K(t,s,u(s))ds$$

becomes

```
u = f(t) + int{K(t,t',u)}
```

The convolution equation:

$$v(t) = \exp(-t) + \int_0^t e^{-(t-s)^2} v(s)ds$$

would be written as:

```
v(t) = exp(-t) + int{exp(-t^2)#v}
```

If one wants to solve, say,

$$u(t) = exp(-t) + \int_0^t (t-t')^{-mu} K(t,t',u(t'))dt'$$

the form is:

```
u(t)= exp(-t) + int[mu]{K(t,t',u}
```

and for convolutions, use the form:

```
u(t)= exp(-t) + int[mu]{w(t)#u}
```

NETWORKS

```
special zip=conv(type,npts,ncon,
                  wgt,root)
```

where `root` is the name of a variable and `wgt` is a table, produces an array `zip` with `npts`:

$$\text{zip}[i] = \sum_{j=-\text{ncon}}^{\text{ncon}} \text{wgt}[j+ncon]\text{root}[i+j]$$

The `sparse` network has the syntax:

```
special zip=sparse(npts,ncon,wgt,
index,root)
```

where `wgt` and `index` are tables with at least `npts * ncon` entries. The array `index` returns the indices of the offsets to with which to connect and the array `wgt` is the coupling strength. The return is

```
zip[i] = sum(j=0;j<ncon)
     w[i*ncon+j]*root[k]
k = index[i*ncon+j]
```

The other two types of networks allow more complicated interactions:

```
special zip=fconv(type,npts,ncon,
               wgt,root1,root2,f)
```

evaluates as

```
zip[i]=sum(j=-ncon;j=ncon)
 wgt[ncon+j]*f(root1[i+j],root2[i])
```

and

```
special zip=fsparse(npts,ncon,wgt,
              index,root1,root2,f)
```

evaluates as

```
zip[i]=sum(j=0;j<ncon)
wgt[ncon*i+j]*f(root1[k],root2[i])
k = index[i*ncon+j]
```

OPTIONS

The format for changing the options is:

```
@ name1=value1, name2=value2, ...
```

where `name` is one of the following and `value` is either an integer, floating point, or string. (All names can be upper or lower case). The first four options *can only be set outside the program.* They are:

- MAXSTOR=`integer` sets the total number of time steps that will be kept in memory. The default is 5000. If you want to perform very long integrations change this to some large number.
- BACK= {`Black,White`} sets the background to black or white.
- SMALL=`fontname` where `fontname` is some font available to your X-server. This sets the "small" font which is used in the Data Browser and in some other windows.
- BIG=`fontname` sets the font for all the menus and popups.
- SMC={0,...,10} sets the stable manifold color
- UMC={0,...,10} sets the unstable manifold color
- XNC={0,...,10} sets the X-nullcline color
- YNC={0,...,10} sets the Y-nullcline color

The remaining options can be set from within the program. They are

- LT=`int` sets the linetype. It should be less than 2 and greater than -6.
- SEED=`int` sets the random number generator seed.
- XP=name sets the name of the variable to plot on the x-axis. The default is `T`, the time-variable.
- YP=name sets the name of the variable on the y-axis.
- ZP=name sets the name of the variable on the z-axis (if the plot is 3D.)
- NPLOT=`int` tells XPP how many plots will be in the opening screen.
- XP2=name,YP2=name,ZP2=name tells XPP the variables on the axes of the second curve; XP8 etc are for the 8th plot. Up to 8 total plots can be specified on opening. They will be given different colors.
- AXES={`2,3`} determine whether a 2D or 3D plot will be displayed.
- TOTAL=value sets the total amount of time to integrate the equations (default is 20).
- DT=value sets the time step for the integrator (default is 0.05).
- NJMP=`integer` tells XPP how frequently to output the solution to the ODE. The default is 1, which means at each integration step.
- T0=value sets the starting time (default is 0).
- TRANS=value tells XPP to integrate until `T=TRANS` and then start plotting solutions (default is 0.)
- NMESH=`integer` sets the mesh size for computing nullclines (default is 40).
- METH={ `discrete, euler, modeuler, rungekutta, adams, gear, volterra, backeul, qualrk, stiff, cvode, 5dp, 83dp, 2rb, ymp`} sets the integration method (default is Runge-Kutta.)
- DTMIN=value sets the minimum allowable timestep for the Gear integrator.
- DTMAX=value sets the maximum allowable timestep for the Gear integrator
- VMAXPTS=value sets the number of points maintained in for the Volterra integral solver. The default is 4000.
- JAC_EPS=value, NEWT_TOL=value,
  NEWT_ITER=value set parameters for the root finders.
- ATOLER=value sets the absolute tolerance for CVODE.
- TOLER=value sets the error tolerance for the Gear, adaptive RK, and stiff integrators. It is the relative tolerance for CVODE.
- BOUND=value sets the maximum bound any plotted variable can reach in magnitude. If any plottable quantity exceeds this, the integrator will halt with a warning. The program will not stop however (default is 100.)
- DELAY=value sets the maximum delay allowed in the integration (default is 0.)
- PHI=value,THETA=value set the angles for the three-dimensional plots.
- XLO=value, YLO=value, XHI=value, YHI=value set the limits for two-dimensional plots (defaults are 0,-2,20,2 respectively.) Note that for three-dimensional plots, the plot is scaled to a cube with vertices that are ± 1 and this cube is rotated and projected onto the plane so setting these to ± 2 works well for 3D plots.

- XMAX=value, XMIN=value, YMAX=value, YMIN=value, ZMAX=value, ZMIN=value set the scaling for three-d plots.
- OUTPUT=filename sets the filename to which you want to write for "silent" integration. The default is "output.dat".
- POIMAP={ `section,maxmin`} sets up a Poincare map for either sections of a variable or the extrema.
- POIVAR=name sets the variable name whose section you are interested in finding.
- POIPLN=value is the value of the section; it is a floating point.
- POISGN={ `1, -1, 0` } determines the direction of the section.
- POISTOP=1 means to stop the integration when the section is reached.
- RANGE=1 means that you want to run a range integration (in batch mode).
- RANGEOVER=name, RANGESTEP=number, RANGELOW=number, RANGEHIGH=number, RANGERESET=`Yes,No`, RANGEOLDIC=`Yes,No` all correspond to the entries in the range integration option.
- TOR_PER=value, defined the period for a toroidal phasespace and tellx XPP that there will be some variables on the circle.
- FOLD=name, tells XPP that the variable ¡name¿ is to be considered modulo the period. You can repeat this for many variables.
- AUTO-stuff. The following AUTO-specific variables can also be set: `NTST, NMAX, NPR, DSMIN, DSMAX, DS, PARMIN, PARMAX, NORMMIN, NORMMAX, AUTOXMIN, AUTOXMAX, AUTOYMIN, AUTOYMAX,`
  `AUTOVAR`. The last is the variable to plot on the y-axis. The x-axis variable is always the first parameter in the ODE file unless you change it within AUTO.

COLOR MEANING 0-Black/White; 1-Red; 2-Red Orange; 3-Orange; 4-Yellow Orange; 5-Yellow; 6-Yellow Green; 7-Green; 8-Blue Green; 9-Blue; 10-Purple.

KEYWORDS You should be aware of the following keywords that should not be used in your ODE files for anything other than their meaning here.

```
sin cos tan atan atan2 sinh cosh tanh
exp delay ln log log10 t pi if then else
asin acos heav sign ceil flr ran abs del\_shft
max min normal besselj bessely erf erfc
arg1 ... arg9  @ \$ + - / * ^ ** shift
| > < == >= <= != not \# int sum of i'
```

These are mainly self-explanatory. The nonobvious ones are:

- `heav(arg1)` the step function, zero if `arg1<0` and 1 otherwise.
- `sign(arg)` which is the sign of the argument (zero has sign 0)
- `ran(arg)` produces a uniformly distributed random number between 0 and `arg`.

- `besselj, bessely` take two arguments, n, x and return respectively, $J_n(x)$ and $Y_n(x)$, the Bessel functions.
- `erf(x), erfc(x)` are the error function and the complementary function.
- `normal(arg1,arg2)` produces a normally distributed random number with mean `arg1` and variance `arg2`.
- `max(arg1,arg2)` produces the maximum of the two arguments and `min` is the minimum of them.
- `if(<exp1>)then(<exp2>)else(<exp3>)` evaluates `<exp1>` If it is nonzero it evaluates to `<exp2>` otherwise it is `<exp3>`. E.g. `if(x>1)then(ln(x))else(x-1)` will lead to `ln(2)` if `x=2` and `-1 if x=0.`
- `delay(<var>,<exp>)` returns variable `<var>` delayed by the result of evaluating `<exp>`. In order to use the delay you must inform the program of the maximal possible delay so it can allocate storage.
- `del_shft(<var>,<shft>,<delay>).` This operator combines the `delay` and the `shift` operators and returns the value of the variable `<var>` shifted by `<shft>` at the delayed time given by `<delay>`.
- `ceil(arg),flr(arg)` are the integer parts of`<arg>` returning the smallest integer greater than and the largest integer less than `<arg>`.
- `t` is the current time in the integration of the differential equation.
- `pi` is π.
- `arg1, ..., arg9` are the formal arguments for functions
- `int, #` concern Volterra equations.
- `shift(<var>,<exp>)` This operator evaluates the expression `<exp>` converts it to an integer and then uses this to indirectly address a variable whose address is that of `<var>` plus the integer value of the expression. This is a way to imitate arrays in XPP. For example if you defined the sequence of 5 variables, `u0,u1,u2,u3,u4` one right after another, then `shift(u0,2)` would return the value of `u2.`
- `sum(<ex1>,<ex2>)of(<ex3>)` is a way of summing up things. The expressions `<ex1>,<ex1>` are evaluated and their integer parts are used as the lower and upper limits of the sum. The index of the sum is `i'` so that you cannot have double sums since there is only one index. `<ex3>` is the expression to be summed and will generally involve `i'.` For example `sum(1,10)of(i')` will be evaluated to 55. Another example combines the sum with the shift operator. `sum(0,4)of(shift(u0,i'))` will sum up `u0` and the next four variables that were defined after it.

Bibliography

[1] `http://www.cs.ruu.nl/wais/html/na-dir/sci/nonlinear-faq.html` contains a lengthy list of software packages as well as some comments on the content.

[2] J. M. Aguirregabiria *Dynamics Solver* , `http://tp.lc.ehu.es/jma/ds/ds.html`

[3] P. Blanchard, R.L. Devaney, and G.R. Hall, *Differential Equations*, Brooks Cole, 1998.

[4] R.L. Borrelli and C.S. Coleman,*Differential Equations : A Modeling Perspective*, John Wiley, 1995.

[5] J. Bower and D. Beeman, *The Book of GENESIS*, Telos, 1997.

[6] K. E. Brenan, S. L. Campbell and L. R. Petzold, The Numerical Solution of Initial Value Problems in Differential-Algebraic Equations, Elsevier Science Publishing Co., (1989).

[7] S.D.Cohen and A.C.Hindmarsh, *CVODE 1.0*, `ftp://sunsite.doc.ic.ac.uk/Mirrors/netlib.att.com/netlib/ode`

[8] E. Doedel, *AUTO*, `ftp://ftp.cs.concordia.ca/pub/doedel/auto`.

[9] Ermentrout G.B. and Kopell, N (1991) Multiple pulse interactions and averaging in systems of coupled neural oscillators, J. Math. Biology 29:195-217

[10] Fife, Paul C.; McLeod, J. B. The approach of solutions of nonlinear diffusion equations to travelling front solutions. Arch. Ration. Mech. Anal. 65 (1977), no. 4, 335–361

[11] C. W. Gear, *Numerical Initial Value Problems in Ordinary Differential Equations*, Pretice Hall, 1975.

[12] Daniel T. Gillespie, Exact Stochastic Simulation of Coupled Chemical Reactions J. Phys. Chem., Vol. 81, pp. 2340-2361 (1977)

[13] L Glass and MC Mackey, From Clocks to Chaos,Princeton Univ Press, 1988

[14] J. Guckenheimer, *dstool*, `ftp://cam.cornell.edu/pub/dstool/`

[15] J. Guckenheimer and P. Holmes, *Nonlinear Oscillations, Dynamical Systems, and Bifurcations of Vector Fields*, Springer 1983, New York.

[16] E. Hairer, S. P. Norsett and G. Wanner, *Solving Ordinary Differential Equations I: Nonstiff Problems, 2nd ed.* Springer, 1993.

[17] G.P. Harmer, D. Abbott, Game theory: Losing strategies can win by Parrondo's paradox Nature 402, 864 (1999).

[18] P.E. Kloeden and E. Platen, *Numerical Solution of Stochastic Differential Equations*, Springer, 1992.

[19] H. Kocak, *Differential and Difference Equations through Computer Experiments: PHASER*, Springer 1986.

[20] C Koch and I Segev, Methods of Neuronal Modeling, MIT Press, 1989

[21] Y. Kuznetsov, *Elements of Applied Bifurcation Theory, Second Edition*, Springer, 1998.

[22] Lanchester, F.W., 1908, Aerodonetics, London

[23] TY Li and JA Yorke 1975 Period three implies chaos, Amer Math Monthly, 82:985-992

[24] P. Linz, *Analytical and Numerical Methods for Volterra Equations*, SIAM, 1985.

[25] R. Macey and G. Oster, *MADONNA*, `http://www.berkeleymadonna.com/`

[26] JD Murray, Mathematical Biology, 1988, Springer-Verlag

[27] Paullet,-Joseph; Ermentrout,-Bard; Troy,-William, The existence of spiral waves in an oscillatory reaction-diffusion system. SIAM-J.-Appl.-Math. 54 (1994), no. 5, 1386–1401.

[28] W.H. Press, S.A. Teukolsky, B.P. Flannnery, and W.T. Vetterling, *Numerical Recipes in C, Second Edition*, Cambridge, 1992.

[29] Werner Rheinboldt, MANPAK `www.netlib.org/contin/manpak`

[30] Shaw, Robert, The Dripping Faucet as a Model Chaotic System, Aerial Press, Santa Cruz, USA, 1984

[31] L. F. Shampine and M.W. Reichelt, The MATLAB ODE Suite, SIAM Journal on Scientific Computing, 18-1, 1997.

[32] S. Strogatz, *Nonlinear Dynamics and Chaos : with Applications in Physics, Biology, Chemistry, and Engineering*, Addison-Wesley, 1994.

[33] C. van Vreeswijk, L.F. Abbott, and B. Ermentrout, *J. Computational Neuroscience*,1:313-321, 1994.

[34] West, B., Strogatz, S., McDill, JM, Cantwell, J. 1997, Interactive Differential Equations, Chapt 19, Addison Wesley Interactive, US

[35] T.L. Williams and G. Bowtell, *J Computational Neuroscience*,4:47-55, 1997.

[36] Winfree, A.T. (1967) Biological rhythms and the behavior of populations of coupled oscillators, J. Theor. Biol. 16:15-42.